GNAP
Internet-Draft
Intended status: Standards Track
Expires: 7 September 2022

J. Richer, Ed.
Bespoke Engineering
A. Parecki
Okta
F. Imbault
acert.io

6 March 2022

Grant Negotiation and Authorization Protocol
draft-ietf-gnap-core-protocol-09

Abstract

GNAP defines a mechanism for delegating authorization to a piece of software, and conveying that delegation to the software. This delegation can include access to a set of APIs as well as information passed directly to the software.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This protocol allows a piece of software, the client instance, to request delegated authorization to resource servers and to request direct information. This delegation is facilitated by an authorization server usually on behalf of a resource owner. The end user operating the software may interact with the authorization server to authenticate, provide consent, and authorize the request.

The process by which the delegation happens is known as a grant, and GNAP allows for the negotiation of the grant process over time by multiple parties acting in distinct roles.

This specification focuses on the portions of the delegation process facing the client instance. In particular, this specification defines interoperable methods for a client instance to request, negotiate, and receive access to information facilitated by the authorization server. This specification also discusses discovery mechanisms for the client instance to configure itself dynamically. The means for an authorization server and resource server to interoperate are discussed in the companion document, [I-D.ietf-gnap-resource-servers].

The focus of this protocol is to provide interoperability between the different parties acting in each role, and is not to specify implementation details of each. Where appropriate, GNAP may make recommendations about internal implementation details, but these recommendations are to ensure the security of the overall deployment rather than to be prescriptive in the implementation.

This protocol solves many of the same use cases as OAuth 2.0 [RFC6749], OpenID Connect [OIDC], and the family of protocols that have grown up around that ecosystem. However, GNAP is not an extension of OAuth 2.0 and is not intended to be directly compatible with OAuth 2.0. GNAP seeks to provide functionality and solve use cases that OAuth 2.0 cannot easily or cleanly address. Appendix B further details the protocol rationale compared to OAuth 2.0. GNAP and OAuth 2.0 will likely exist in parallel for many deployments, and considerations have been taken to facilitate the mapping and transition from legacy systems to GNAP. Some examples of these can be found in Appendix D.5.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document contains non-normative examples of partial and complete HTTP messages, JSON structures, URIs, query components, keys, and other elements. Whenever possible, the document uses URI as a generic term, since it aligns with [RFC3986] recommendations and matches better with the intent that the identifier may be reachable through various/generic means (compared to URLs). Some examples use a single trailing backslash \ to indicate line wrapping for long values, as per [RFC8792]. The \ character and leading spaces on wrapped lines are not part of the value.

1.2. Roles

The parties in GNAP perform actions under different roles. Roles are defined by the actions taken and the expectations leveraged on the role by the overall protocol.

```
   +-------------+            +------------+
   |             |            |            |
   |Authorization|            |  Resource  |
   |   Server    |            |   Server   |
   |             |<--+   +--->|            |
   +-------------+   |   |    +------------+
           +         |   |
           +         |   |
           +         |   |
           +         |   |
           +         |   |
           +      +----------+
           +      |  Client  |
           +      | Instance |
           +      +----------+
           +           +
           +           +
           +           +
   +-----------+       +     +------------+
   |           |       + + + |            |
   |  Resource |             |    End     |
   |   Owner   | ~ ~ ~ ~ ~ ~ |    User    |
   |           |             |            |
   +-----------+             +------------+
```

Legend

```
   + + + indicates interaction between a human and computer
   ----- indicates interaction between two pieces of software
   ~ ~ ~ indicates a potential equivalence or out-of-band
             communication between roles
```

Authorization Server (AS) server that grants delegated privileges to a particular instance of client software in the form of access tokens or other information (such as subject information).

Client application that consumes resources from one or several RSs, possibly requiring access privileges from one or several ASs. The client is operated by the end user or it runs autonomously on behalf of a resource owner.

Example: a client can be a mobile application, a web application, etc.

Note: this specification differentiates between a specific instance (the client instance, identified by its unique key) and the software running the instance (the client software). For some kinds of client software, there could be many instances of that software, each instance with a different key.

Resource Server (RS) server that provides operations on protected resources, where operations require a valid access token issued by an AS.

Resource Owner (RO) subject entity that may grant or deny operations on resources it has authority upon.

Note: the act of granting or denying an operation may be manual (i.e. through an interaction with a physical person) or automatic (i.e. through predefined organizational rules).

End user natural person that operates a client instance.

Note: that natural person may or may not be the same entity as the RO.

The design of GNAP does not assume any one deployment architecture, but instead attempts to define roles that can be fulfilled in a number of different ways for different use cases. As long as a given role fulfills all of its obligations and behaviors as defined by the protocol, GNAP does not make additional requirements on its structure or setup.

Multiple roles can be fulfilled by the same party, and a given party can switch roles in different instances of the protocol. For example, the RO and end user in many instances are the same person, where a user is authorizing the client instance to act on their own behalf at the RS. In this case, one party fulfills both of the RO and end-user roles, but the roles themselves are still defined separately from each other to allow for other use cases where they are fulfilled by different parties.

For another example, in some complex scenarios, an RS receiving requests from one client instance can act as a client instance for a downstream secondary RS in order to fulfill the original request. In this case, one piece of software is both an RS and a client instance from different perspectives, and it fulfills these roles separately as far as the overall protocol is concerned.

A single role need not be deployed as a monolithic service. For example, a client instance could have components that are installed on the end user's device as well as a back-end system that it communicates with. If both of these components participate in the delegation protocol, they are both considered part of the client instance. If there are several copies of the client software that run separately but all share the same key material, such as a deployed cluster, then this cluster is considered a single client instance.

In these cases, the distinct components of what is considered a GNAP client instance may use any number of different communication mechanisms between them, all of which would be considered an implementation detail of the client instances and out of scope of GNAP.

For another example, an AS could likewise be built out of many constituent components in a distributed architecture. The component that the client instance calls directly could be different from the component that the RO interacts with to drive consent, since API calls and user interaction have different security considerations in many environments. Furthermore, the AS could need to collect identity claims about the RO from one system that deals with user attributes while generating access tokens at another system that deals with security rights. From the perspective of GNAP, all of these are pieces of the AS and together fulfill the role of the AS as defined by the protocol. These pieces may have their own internal communications mechanisms which are considered out of scope of GNAP.

1.3. Elements

In addition to the roles above, the protocol also involves several elements that are acted upon by the roles throughout the process.

Attribute characteristics related to a subject.

Access Token a data artifact representing a set of rights and/or attributes.

Note: an access token can be first issued to an client instance (requiring authorization by the RO) and subsequently rotated.

Grant (verb): to permit an instance of client software to receive some attributes at a specific time and valid for a specific duration and/or to exercise some set of delegated rights to access a protected resource (noun): the act of granting.

Privilege right or attribute associated with a subject.

Note: the RO defines and maintains the rights and attributes associated to the protected resource, and might temporarily delegate some set of those privileges to an end user. This process is refered to as privilege delegation.

Protected Resource protected API (Application Programming Interface) served by an RS and that can be accessed by a client, if and only if a valid access token is provided.

Note: to avoid complex sentences, the specification document may simply refer to "resource" instead of "protected resource".

Right ability given to a subject to perform a given operation on a resource under the control of an RS.

Subject person, organization or device. It decides whether and under which conditions its attributes can be disclosed to other parties.

Subject Information statement asserted by an AS about a subject.

1.4. Trust relationships

GNAP defines its trust objective as: "the RO trusts the AS to ensure access validation and delegation of protected resources to end users, through third party clients."

This trust objective can be decomposed into trust relationships between software elements and roles, especially the pairs end user/ RO, end user/client, client/AS, RS/RO, AS/RO, AS/RS. Trust of an agent by its pair can exist if the pair is informed that the agent has made a promise to follow the protocol in the past (e.g. pre-registration, uncompromised cryptographic components) or if the pair is able to infer by indirect means that the agent has made such a promise (e.g. a compliant client request). Each agent defines its own valuation function of promises given or received. Examples of such valuations can be the benefits from interacting with other agents (e.g. safety in client access, interoperability with identity standards), the cost of following the protocol (including its security and privacy requirements and recommendations), a ranking of promise importance (e.g. a policy decision made by the AS), the assessment of one's vulnerability or risk of not being able to defend against threats, etc. Those valuations may depend on the context of the request. For instance, the AS may decide to either take into account or discard hints provided by the client, the RS may refuse bearer tokens, etc. depending on the specific case in which GNAP is used. Some promises can be conditional of some previous interactions (e.g. repeated requests).

Looking back on each trust relationship:

* end user/RO: this relationship exists only when the end user and the RO are different, in which case the end user needs some out of band mechanism of getting the RO consent (see Section 4). GNAP generally assumes that humans can be authenticated thanks to identity protocols (for instance, through an id_token assertion in Section 2.2).

* end user/client: the client acts as a user agent. Depending on the technology used (browser, SPA, mobile application, IoT device, etc.), some interactions may or may not be possible (as described in Section 2.5.1). Client developers promise to implement requirements and generally some recommendations or best practices, so that the end users may confidently use their software. However, end users might also be facing some attacker's client software, without even realizing it.

* end user/AS: when the client supports it (see Section 3.3), the end user gets to interact with front-channel URIs provided by the AS. See Section 12.26 for some considerations in trusting these interactions.

* client/AS: An honest AS may be facing an attacker's client (as discussed just above), or the reverse, and GNAP aims at making common attacks impractical. The core specification makes access tokens opaque to the client and defines the request/response scheme in detail, therefore avoiding extra trust hypotheses from this critical piece of software. Yet the AS may further define cryptographic attestations or optional rules to simplify the access of clients it already trusts, due to past behavior or organizational policies (see Section 2.3).

* RS/RO: the RS promises it protects its resources from unauthorized access, and only accepts valid access tokens issued by a trusted AS. In case tokens are key bound, proper validation is expected from the RS.

* AS/RO: the AS is expected to follow the decisions made by the RO, either through interactive consent requests, repeated interactions or automated rules (as described in Section 1.5). Privacy considerations aim to reduce the risk of an honest but too curious AS, or the consequences of an unexpected user data exposure.

* AS/RS: the AS promises to issue valid access tokens to legitimate client requests (i.e. after carrying out appropriate due diligence, as defined in the GNAP protocol). Some optional configurations are covered by [I-D.ietf-gnap-resource-servers].

A global assumption made by GNAP is that authorization requests are security and privacy sensitive, and appropriate measures are respectively detailed in Section 12 and Section 13.

A formal trust model is out of scope of this specification, but might be carried out thanks to [promise-theory].

1.5. Sequences

GNAP can be used in a variety of ways to allow the core delegation process to take place. Many portions of this process are conditionally present depending on the context of the deployments, and not every step in this overview will happen in all circumstances.

Note that a connection between roles in this process does not necessarily indicate that a specific protocol message is sent across the wire between the components fulfilling the roles in question, or that a particular step is required every time. For example, for a client instance interested in only getting subject information directly, and not calling an RS, all steps involving the RS below do not apply.

In some circumstances, the information needed at a given stage is communicated out of band or is preconfigured between the components or entities performing the roles. For example, one entity can fulfill multiple roles, and so explicit communication between the roles is not necessary within the protocol flow. Additionally some components may not be involved in all use cases. For example, a client instance could be calling the AS just to get direct user information and have no need to get an access token to call an RS.

1.5.1. Overall Protocol Sequence

The following diagram provides a general overview of GNAP, including many different optional phases and connections. The diagrams in the following sections provide views of GNAP under more specific circumstances.

```
       +------------+         +------------+
       | End user   | ~ ~ ~ ~ |  Resource  |
       |            |         | Owner (RO) |
       +------------+         +------------+
             +                       +
             +                       +
             (A)                     (B)
             +                       +
             +                       +
       +--------+                    +          +------------+
       | Client | (1)                +          |  Resource  |
       |Instance|                    +          |   Server   |
       |        |       +---------------+       |    (RS)    |
       |        |--(2)->| Authorization |       |            |
       |        |<-(3)--|     Server    |       |            |
       |        |       |      (AS)     |       |            |
       |        |--(4)->|               |       |            |
       |        |<-(5)--|               |       |            |
       |        |--------------(6)------------->|            |
       |        |       |               |   (7) |            |
       |        |<-------------(8)------------->|            |
       |        |--(9)->|               |       |            |
       |        |<-(10)-|               |       |            |
       |        |--------------(11)------------>|            |
       |        |       |               |  (12) |            |
       |        |-(13)->|               |       |            |
       |        |       |               |       |            |
       +--------+       +---------------+       +------------+

   Legend
   + + + indicates a possible interaction with a human
   ----- indicates an interaction between protocol roles
   ~ ~ ~ indicates a potential equivalence or out-of-band
           communication between roles
```

* (A) The end user interacts with the client instance to indicate a need for resources on behalf of the RO. This could identify the RS the client instance needs to call, the resources needed, or the RO that is needed to approve the request. Note that the RO and end user are often the same entity in practice, but GNAP makes no general assumption that they are.

* (1) The client instance determines what access is needed and which AS to approach for access. Note that for most situations, the client instance is pre-configured with which AS to talk to and which kinds of access it needs, but some more dynamic processes are discussed in Section 9.1.

* (2) The client instance requests access at the AS (Section 2).

* (3) The AS processes the request and determines what is needed to fulfill the request. (See Section 4.) The AS sends its response to the client instance (Section 3).

* (B) If interaction is required, the AS interacts with the RO (Section 4) to gather authorization. The interactive component of the AS can function using a variety of possible mechanisms including web page redirects, applications, challenge/response protocols, or other methods. The RO approves the request for the client instance being operated by the end user. Note that the RO and end user are often the same entity in practice, and many of GNAP's interaction methods allow the client instance to facilitate the end user interacting with the AS in order to fulfill the role of the RO.

* (4) The client instance continues the grant at the AS (Section 5).

* (5) If the AS determines that access can be granted, it returns a response to the client instance (Section 3) including an access token (Section 3.2) for calling the RS and any directly returned information (Section 3.4) about the RO.

* (6) The client instance uses the access token (Section 7.2) to call the RS.

* (7) The RS determines if the token is sufficient for the request by examining the token. The means of the RS determining this access are out of scope of this specification, but some options are discussed in [I-D.ietf-gnap-resource-servers].

* (8) The client instance calls the RS (Section 7.2) using the access token until the RS or client instance determine that the token is no longer valid.

* (9) When the token no longer works, the client instance fetches an updated access token (Section 6.1) based on the rights granted in (5).

* (10) The AS issues a new access token (Section 3.2) to the client instance.

* (11) The client instance uses the new access token (Section 7.2) to call the RS.

* (12) The RS determines if the new token is sufficient for the request. The means of the RS determining this access are out of scope of this specification, but some options are discussed in [I-D.ietf-gnap-resource-servers].

* (13) The client instance disposes of the token (Section 6.2) once the client instance has completed its access of the RS and no longer needs the token.

The following sections and Appendix D contain specific guidance on how to use GNAP in different situations and deployments. For example, it is possible for the client instance to never request an access token and never call an RS, just as it is possible for there not to be a user involved in the delegation process.

1.5.2. Redirect-based Interaction

In this example flow, the client instance is a web application that wants access to resources on behalf of the current user, who acts as both the end user and the resource owner (RO). Since the client instance is capable of directing the user to an arbitrary URI and receiving responses from the user's browser, interaction here is handled through front-channel redirects using the user's browser. The redirection URI used for interaction is a service hosted by the AS in this example. The client instance uses a persistent session with the user to ensure the same user that is starting the interaction is the user that returns from the interaction.

```
 +--------+                                  +--------+         +------+
 | Client |                                  |   AS   |         | User |
 |Instance|                                  |        |         |      |
 |        |< (1) + Start Session + + + + + + + + + + + + + + + + +|      |
 |        |                                  |        |         |      |
 |        |--(2)--- Request Access --------->|        |         |      |
 |        |                                  |        |         |      |
 |        |<-(3)-- Interaction Needed -------|        |         |      |
 |        |                                  |        |         |      |
 |        |+ (4) + Redirect for Interaction + + + + + + + + + > |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (5) +>|      |
 |        |                                  |        |  AuthN  |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (6) +>|      |
 |        |                                  |        |  AuthZ  |      |
 |        |                                  |        |         |      |
 |        |< (7) + Redirect for Continuation + + + + + + + + + +|      |
 |        |                                  |        |         +------+
 |        |--(8)--- Continue Request ------->|        |
 |        |                                  |        |
 |        |<-(9)----- Grant Access ----------|        |
 |        |                                  |        |
 |        |                                  |        |     +--------+
 |        |--(10)-- Access API ---------------------------->|   RS   |
 |        |                                  |        |     |        |
 |        |<-(11)-- API Response ---------------------------|        |
 |        |                                  |        |     +--------+
 +--------+                                  +--------+
```

1. The client instance establishes a verifiable session to the user, in the role of the end user.

2. The client instance requests access to the resource (Section 2). The client instance indicates that it can redirect to an arbitrary URI (Section 2.5.1.1) and receive a redirect from the browser (Section 2.5.2.1). The client instance stores verification information for its redirect in the session created in (1).

3. The AS determines that interaction is needed and responds (Section 3) with a URI to send the user to (Section 3.3.1) and information needed to verify the redirect (Section 3.3.5) in (7). The AS also includes information the client instance will need to continue the request (Section 3.1) in (8). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), and (8).

4. The client instance stores the verification and continuation information from (3) in the session from (1). The client instance then redirects the user to the URI (Section 4.1.1) given by the AS in (3). The user's browser loads the interaction redirect URI. The AS loads the pending request based on the incoming URI generated in (3).

5. The user authenticates at the AS, taking on the role of the RO.

6. As the RO, the user authorizes the pending request from the client instance.

7. When the AS is done interacting with the user, the AS redirects the user back (Section 4.2.1) to the client instance using the redirect URI provided in (2). The redirect URI is augmented with an interaction reference that the AS associates with the ongoing request created in (2) and referenced in (4). The redirect URI is also augmented with a hash of the security information provided in (2) and (3). The client instance loads the verification information from (2) and (3) from the session created in (1). The client instance calculates a hash (Section 4.2.3) based on this information and continues only if the hash validates. Note that the client instance needs to ensure that the parameters for the incoming request match those that it is expecting from the session created in (1). The client instance also needs to be prepared for the end user never being returned to the client instance and handle timeouts appropriately.

8. The client instance loads the continuation information from (3) and sends the interaction reference from (7) in a request to continue the request (Section 5.1). The AS validates the interaction reference ensuring that the reference is associated with the request being continued.

9. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

10. The client instance uses the access token (Section 7.2) to call the RS.

11. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.1.

1.5.3. User-code Interaction

In this example flow, the client instance is a device that is capable of presenting a short, human-readable code to the user and directing the user to enter that code at a known URI. The URI the user enters the code at is an interactive service hosted by the AS in this example. The client instance is not capable of presenting an arbitrary URI to the user, nor is it capable of accepting incoming HTTP requests from the user's browser. The client instance polls the AS while it is waiting for the RO to authorize the request. The user's interaction is assumed to occur on a secondary device. In this example it is assumed that the user is both the end user and RO, though the user is not assumed to be interacting with the client instance through the same web browser used for interaction at the AS.

```
 +--------+                                  +--------+         +------+
 | Client |                                  |   AS   |         | User |
 |Instance|--(1)--- Request Access --------->|        |         |      |
 |        |                                  |        |         |      |
 |        |<-(2)-- Interaction Needed -------|        |         |      |
 |        |                                  |        |         |      |
 |        |+ (3) + + Display User Code + + + + + + + + + + + + >|      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (4) + |      |
 |        |                                  |        |Open URI |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (5) +>|      |
 |        |                                  |        |  AuthN  |      |
 |        |--(9)--- Continue Request (A) --->|        |         |      |
 |        |                                  |        |<+ (6) +>|      |
 |        |<-(10)- Not Yet Granted (Wait) ---|        |  Code   |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (7) +>|      |
 |        |                                  |        |  AuthZ  |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (8) +>|      |
 |        |                                  |        |Completed|      |
 |        |                                  |        |         |      |
 |        |--(11)-- Continue Request (B) --->|        |         +------+
 |        |                                  |        |
 |        |<-(12)----- Grant Access ---------|        |
 |        |                                  |        |
 |        |                                  |        |     +--------+
 |        |--(13)-- Access API ---------------------------->|   RS   |
 |        |                                  |        |     |        |
 |        |<-(14)-- API Response ---------------------------|        |
 |        |                                  |        |     +--------+
 +--------+                                  +--------+
```

1. The client instance requests access to the resource (Section 2). The client instance indicates that it can display a user code (Section 2.5.1.3).

2. The AS determines that interaction is needed and responds (Section 3) with a user code to communicate to the user (Section 3.3.3). This could optionally include a URI to direct the user to, but this URI should be static and so could be configured in the client instance's documentation. The AS also includes information the client instance will need to continue the request (Section 3.1) in (8) and (10). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), (8), and (10).

3. The client instance stores the continuation information from (2) for use in (8) and (10). The client instance then communicates the code to the user (Section 4.1.2) given by the AS in (2).

4. The users directs their browser to the user code URI. This URI is stable and can be communicated via the client software's documentation, the AS documentation, or the client software itself. Since it is assumed that the RO will interact with the AS through a secondary device, the client instance does not provide a mechanism to launch the RO's browser at this URI.

5. The end user authenticates at the AS, taking on the role of the RO.

6. The RO enters the code communicated in (3) to the AS. The AS validates this code against a current request in process.

7. As the RO, the user authorizes the pending request from the client instance.

8. When the AS is done interacting with the user, the AS indicates to the RO that the request has been completed.

9. Meanwhile, the client instance loads the continuation information stored at (3) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.

10. If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polled continuation requests. This response can include updated continuation information as well as information regarding how long the client instance should wait before calling again. The

client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.

11. The client instance continues to poll the AS (Section 5.2) with the new continuation information in (9).

12. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

13. The client instance uses the access token (Section 7.2) to call the RS.

14. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.2.

1.5.4. Asynchronous Authorization

In this example flow, the end user and RO roles are fulfilled by different parties, and the RO does not interact with the client instance. The AS reaches out asynchronously to the RO during the request process to gather the RO's authorization for the client instance's request. The client instance polls the AS while it is waiting for the RO to authorize the request.

```
 +--------+                                  +--------+         +------+
 | Client |                                  |   AS   |         |  RO  |
 |Instance|--(1)--- Request Access --------->|        |         |      |
 |        |                                  |        |         |      |
 |        |<-(2)-- Not Yet Granted (Wait) ---|        |         |      |
 |        |                                  |        |<+ (3) +>|      |
 |        |                                  |        |  AuthN  |      |
 |        |--(6)--- Continue Request (A) --->|        |         |      |
 |        |                                  |        |<+ (4) +>|      |
 |        |<-(7)-- Not Yet Granted (Wait) ---|        |  AuthZ  |      |
 |        |                                  |        |         |      |
 |        |                                  |        |<+ (5) +>|      |
 |        |                                  |        |Completed|      |
 |        |                                  |        |         |      |
 |        |--(8)--- Continue Request (B) --->|        |         +------+
 |        |                                  |        |
 |        |<-(9)------ Grant Access ---------|        |
 |        |                                  |        |
 |        |                                  |        |     +--------+
 |        |--(10)-- Access API ---------------------------->|   RS   |
 |        |                                  |        |     |        |
 |        |<-(11)-- API Response ----------------------------|        |
 |        |                                  |        |     +--------+
 +--------+                                  +--------+
```

1. The client instance requests access to the resource (Section 2). The client instance does not send any interaction modes to the server, indicating that it does not expect to interact with the RO. The client instance can also signal which RO it requires authorization from, if known, by using the user request section (Section 2.4).

2. The AS determines that interaction is needed, but the client instance cannot interact with the RO. The AS responds (Section 3) with the information the client instance will need to continue the request (Section 3.1) in (6) and (8), including a signal that the client instance should wait before checking the status of the request again. The AS associates this continuation information with an ongoing request that will be referenced in (3), (4), (5), (6), and (8).

3. The AS determines which RO to contact based on the request in (1), through a combination of the user request (Section 2.4), the resources request (Section 2.1), and other policy information. The AS contacts the RO and authenticates them.

4. The RO authorizes the pending request from the client instance.

5.   When the AS is done interacting with the RO, the AS indicates to the RO that the request has been completed.

6.   Meanwhile, the client instance loads the continuation information stored at (2) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.

7.   If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polling. This response can include refreshed credentials as well as information regarding how long the client instance should wait before calling again. The client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.

8.   The client instance continues to poll the AS (Section 5.2) with the new continuation information from (7).

9.   If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

10. The client instance uses the access token (Section 7.2) to call the RS.

11. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.4.

1.5.5. Software-only Authorization

In this example flow, the AS policy allows the client instance to make a call on its own behalf, without the need for an RO to be involved at runtime to approve the decision. Since there is no explicit RO, the client instance does not interact with an RO.

```
   +--------+                                  +--------+
   | Client |                                  |   AS   |
   |Instance|--(1)--- Request Access --->|        |
   |        |                                  |        |
   |        |<-(2)---- Grant Access -----|        |
   |        |                                  |        |  +--------+
   |        |--(3)--- Access API ------------------->|   RS   |
   |        |                                  |        |  |        |
   |        |<-(4)--- API Response ------------------|        |
   |        |                                  |        |  +--------+
   +--------+                                  +--------+
```

1. The client instance requests access to the resource (Section 2). The client instance does not send any interaction modes to the server.

2. The AS determines that the request has been authorized, the AS grants access to the resource in the form of access tokens (Section 3.2) to the client instance. Note that direct subject information (Section 3.4) is not generally applicable in this use case, as there is no user involved.

3. The client instance uses the access token (Section 7.2) to call the RS.

4. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.3.

1.5.6. Refreshing an Expired Access Token

In this example flow, the client instance receives an access token to access a resource server through some valid GNAP process. The client instance uses that token at the RS for some time, but eventually the access token expires. The client instance then gets a new access token by rotating the expired access token at the AS using the token's management URI.

```
   +--------+                                          +--------+
   | Client |                                          |   AS   |
   |Instance|--(1)--- Request Access ----------------->|        |
   |        |                                          |        |
   |        |<-(2)--- Grant Access --------------------|        |
   |        |                                          |        |
   |        |                             +--------+   |        |
   |        |--(3)--- Access Resource --->|   RS   |   |        |
   |        |                             |        |   |        |
   |        |<-(4)--- Success Response ---|        |   |        |
   |        |                             |        |   |        |
   |        |                             |        |   |        |
   |        |                             |        |   |        |
   |        |--(5)--- Access Resource --->|        |   |        |
   |        |                             |        |   |        |
   |        |<-(6)--- Error Response -----|        |   |        |
   |        |                             +--------+   |        |
   |        |                                          |        |
   |        |--(7)--- Rotate Token ------------------->|        |
   |        |                                          |        |
   |        |<-(8)--- Rotated Token -------------------|        |
   |        |                                          |        |
   +--------+                                          +--------+
```

1. The client instance requests access to the resource (Section 2).

2. The AS grants access to the resource (Section 3) with an access token (Section 3.2) usable at the RS. The access token response includes a token management URI.

3. The client instance uses the access token (Section 7.2) to call the RS.

4. The RS validates the access token and returns an appropriate response for the API.

5. Time passes and the client instance uses the access token to call the RS again.

6. The RS validates the access token and determines that the access token is expired. The RS responds to the client instance with an error.

7. The client instance calls the token management URI returned in (2) to rotate the access token (Section 6.1). The client instance uses the access token (Section 7.2) in this call as well as the appropriate key, see the token rotation section for details.

8. The AS validates the rotation request including the signature and keys presented in (5) and returns a new access token (Section 3.2.1). The response includes a new access token and can also include updated token management information, which the client instance will store in place of the values returned in (2).

1.5.7. Requesting User Information

In this scenario, the client instance does not call an RS and does not request an access token. Instead, the client instance only requests and is returned direct subject information (Section 3.4). Many different interaction modes can be used in this scenario, so these are shown only in the abstract as functions of the AS here.

```
+--------+                                  +--------+         +------+
| Client |                                  |   AS   |         | User |
|Instance|                                  |        |         |      |
|        |--(1)--- Request Access --------->|        |         |      |
|        |                                  |        |         |      |
|        |<-(2)-- Interaction Needed -------|        |         |      |
|        |                                  |        |         |      |
|        |+ (3) + Facilitate Interaction + + + + + + + + + + > |      |
|        |                                  |        |         |      |
|        |                                  |        |<+ (4) +>|      |
|        |                                  |        |  AuthN  |      |
|        |                                  |        |         |      |
|        |                                  |        |<+ (5) +>|      |
|        |                                  |        |  AuthZ  |      |
|        |                                  |        |         |      |
|        |< (6) + Signal Continuation + + + + + + + + + + + + +|      |
|        |                                  |        |         +------+
|        |--(7)--- Continue Request ------->|        |
|        |                                  |        |
|        |<-(8)----- Grant Access ----------|        |
|        |                                  |        |
+--------+                                  +--------+
```

1. The client instance requests access to subject information (Section 2).

2. The AS determines that interaction is needed and responds (Section 3) with appropriate information for facilitating user interaction (Section 3.3).

3. The client instance facilitates the user interacting with the AS (Section 4) as directed in (2).

4. The user authenticates at the AS, taking on the role of the RO.

5. As the RO, the user authorizes the pending request from the client instance.

6. When the AS is done interacting with the user, the AS returns the user to the client instance and signals continuation.

7. The client instance loads the continuation information from (2) and calls the AS to continue the request (Section 5).

8. If the request has been authorized, the AS grants access to the requested direct subject information (Section 3.4) to the client instance. At this stage, the user is generally considered "logged in" to the client instance based on the identifiers and assertions provided by the AS. Note that the AS can restrict the subject information returned and it might not match what the client instance requested, see the section on subject information for details.

2. Requesting Access

To start a request, the client instance sends a JSON [RFC8259] document with an object as its root. Each member of the request object represents a different aspect of the client instance's request. Each field is described in detail in a section below.

access_token (object / array of objects): Describes the rights and properties associated with the requested access token. REQUIRED if requesting an access token. See Section 2.1.

subject (object): Describes the information about the RO that the client instance is requesting to be returned directly in the response from the AS. REQUIRED if requesting subject information. See Section 2.2.

client (object / string): Describes the client instance that is making this request, including the key that the client instance will use to protect this request and any continuation requests at the AS and any user-facing information about the client instance used in interactions. REQUIRED. See Section 2.3.

user (object / string): Identifies the end user to the AS in a manner that the AS can verify, either directly or by interacting with the end user to determine their status as the RO. OPTIONAL. See Section 2.4.

interact (object): Describes the modes that the client instance

      supports for allowing the RO to interact with the AS and modes for
      the client instance to receive updates when interaction is
      complete. REQUIRED if interaction is supported. See Section 2.5.

Additional members of this request object can be defined by extensions to this protocol as described in Section 2.6.

A non-normative example of a grant request is below:

```
   {
       "access_token": {
           "access": [
               {
                   "type": "photo-api",
                   "actions": [
                       "read",
                       "write",
                       "dolphin"
                   ],
                   "locations": [
                       "https://server.example.net/",
                       "https://resource.local/other"
                   ],
                   "datatypes": [
                       "metadata",
                       "images"
                   ]
               },
               "dolphin-metadata"
           ]
       },
       "client": {
         "display": {
           "name": "My Client Display Name",
           "uri": "https://example.net/client"
         },
         "key": {
           "proof": "httpsig",
           "jwk": {
             "kty": "RSA",
             "e": "AQAB",
             "kid": "xyz-1",
             "alg": "RS256",
             "n": "kOB5rR4Jv0GMeL...."
           }
         }
       },
       "interact": {
           "start": ["redirect"],
           "finish": {
               "method": "redirect",
               "uri": "https://client.example.net/return/123455",
               "nonce": "LKLTI25DK82FX4T4QFZC"
           }
       },
       "subject": {
           "sub_id_formats": ["iss_sub", "opaque"],
           "assertion_formats": ["id_token"]
       }
   }
```

The request and response MUST be sent as a JSON object in the body of the HTTP POST request with Content-Type application/json, unless otherwise specified by the signature mechanism.

The authorization server MUST include the HTTP "Cache-Control" response header field [RFC7234] with a value set to "no-store".

2.1. Requesting Access to Resources

If the client instance is requesting one or more access tokens for the purpose of accessing an API, the client instance MUST include an access_token field. This field MUST be an object (for a single access token (Section 2.1.1)) or an array of these objects (for multiple access tokens (Section 2.1.2)), as described in the following sections.

2.1.1. Requesting a Single Access Token

To request a single access token, the client instance sends an acccess_token object composed of the following fields.

access (array of objects/strings): Describes the rights that the client instance is requesting for one or more access tokens to be used at RS's. REQUIRED. See Section 8.

label (string): A unique name chosen by the client instance to refer to the resulting access token. The value of this field is opaque to the AS. If this field is included in the request, the AS MUST include the same label in the token response (Section 3.2). REQUIRED if used as part of a multiple access token request (Section 2.1.2), OPTIONAL otherwise.

flags (array of strings): A set of flags that indicate desired attributes or behavior to be attached to the access token by the AS. OPTIONAL.

The values of the flags field defined by this specification are as follows:

"bearer": If this flag is included, the access token being requested is a bearer token. If this flag is omitted, the access token is bound to the key used by the client instance in this request (or that key's most recent rotation) and the access token MUST be presented using the same key and proofing method. Methods for presenting bound and bearer access tokens are described in Section 7.2. See Section 12.7 for additional considerations on the use of bearer tokens.

"split": If this flag is included, the client instance is capable of receiving a different number of tokens than specified in the token request (Section 2.1), including receiving multiple access tokens (Section 3.2.2) in response to any single token request (Section 2.1.1) or a different number of access tokens than requested in a multiple access token request (Section 2.1.2). The label fields of the returned additional tokens are chosen by the AS. The client instance MUST be able to tell from the token response where and how it can use each of the access tokens. [[See issue #37 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/37)]]

Flag values MUST NOT be included more than once.

Additional flags can be defined by extensions using a registry TBD (Section 11).

In the following example, the client instance is requesting access to a complex resource described by a pair of access request object.

```
"access_token": {
    "access": [
        {
            "type": "photo-api",
            "actions": [
                "read",
                "write",
                "delete"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        },
        {
            "type": "walrus-access",
            "actions": [
                "foo",
                "bar"
            ],
            "locations": [
                "https://resource.other/"
            ],
            "datatypes": [
                "data",
                "pictures",
                "walrus whiskers"
            ]
        }
    ],
    "label": "token1-23",
    "flags": [ "split" ]
}
```

If access is approved, the resulting access token is valid for the described resource and is bound to the client instance's key (or its most recent rotation). The token is labeled "token1-23" and could be split into multiple access tokens by the AS, if the AS chooses. The token response structure is described in Section 3.2.1.

2.1.2. Requesting Multiple Access Tokens

To request multiple access tokens to be returned in a single response, the client instance sends an array of objects as the value of the access_token parameter. Each object MUST conform to the request format for a single access token request, as specified in requesting a single access token (Section 2.1.1). Additionally, each object in the array MUST include the label field, and all values of these fields MUST be unique within the request. If the client instance does not include a label value for any entry in the array, or the values of the label field are not unique within the array, the AS MUST return an error.

The following non-normative example shows a request for two separate access tokens, token1 and token2.

```
"access_token": [
    {
        "label": "token1",
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "dolphin-metadata"
        ]
    },
    {
        "label": "token2",
        "access": [
            {
                "type": "walrus-access",
                "actions": [
                    "foo",
                    "bar"
                ],
                "locations": [
                    "https://resource.other/"
                ],
                "datatypes": [
                    "data",
                    "pictures",
                    "walrus whiskers"
                ]
            }
        ],
        "flags": [ "bearer" ]
    }
]
```

All approved access requests are returned in the multiple access token response (Section 3.2.2) structure using the values of the label fields in the request.

2.2. Requesting Subject Information

If the client instance is requesting information about the RO from the AS, it sends a subject field as a JSON object. This object MAY contain the following fields (or additional fields defined in a registry TBD (Section 11)).

sub_id_formats (array of strings): An array of subject identifier subject formats requested for the RO, as defined by [I-D.ietf-secevent-subject-identifiers]. REQUIRED if subject identifiers are requested.

assertion_formats (array of strings): An array of requested assertion formats. Possible values include id_token for an [OIDC] ID Token and saml2 for a SAML 2 assertion. Additional assertion formats are defined by a registry TBD (Section 11). REQUIRED if assertions are requested.

```
"subject": {
  "sub_id_formats": [ "iss_sub", "opaque" ],
  "assertion_formats": [ "id_token", "saml2" ]
}
```

The AS can determine the RO's identity and permission for releasing this information through interaction with the RO (Section 4), AS policies, or assertions presented by the client instance (Section 2.4). If this is determined positively, the AS MAY return the RO's information in its response (Section 3.4) as requested.

Subject identifier types requested by the client instance serve only to identify the RO in the context of the AS and can't be used as communication channels by the client instance, as discussed in Section 3.4.

The AS SHOULD NOT re-use subject identifiers for multiple different ROs.

The "formats" and "assertions" request fields are independent of each other, and a returned assertion MAY use a different subject identifier than other assertions and subject identifiers in the response. All subject identifiers and assertions returned MUST refer to the same person.

2.3. Identifying the Client Instance

When sending a non-continuation request to the AS, the client instance MUST identify itself by including the client field of the request and by signing the request as described in Section 7.3. Note that for a continuation request (Section 5), the client instance is identified by its association with the request being continued and so this field is not sent under those circumstances.

When client instance information is sent by value, the client field of the request consists of a JSON object with the following fields.

key (object / string): The public key of the client instance to be used in this request as described in Section 7.1 or a reference to a key as described in Section 7.1.1. REQUIRED.

class_id (string): An identifier string that the AS can use to identify the client software comprising this client instance. The contents and format of this field are up to the AS. OPTIONAL.

display (object): An object containing additional information that the AS MAY display to the RO during interaction, authorization, and management. OPTIONAL.

```
"client": {
    "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-1",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        },
        "cert": "MIIEHDCCAwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
    },
    "class_id": "web-server-1234",
    "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
    }
}
```

Additional fields are defined in a registry TBD (Section 11).

The client instance MUST prove possession of any presented key by the proof mechanism associated with the key in the request. Proof types are defined in a registry TBD (Section 11) and an initial set of methods is described in Section 7.3.

If the same public key is sent by value on different access requests, the AS MUST treat these requests as coming from the same client instance for purposes of identification, authentication, and policy application. If the AS does not know the client instance's public key ahead of time, the AS MAY accept or reject the request based on AS policy, attestations within the client request, and other mechanisms.

[[See issue #44 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/44)]]

The client instance MUST NOT send a symmetric key by value in the request, as doing so would expose the key directly instead of simply proving possession of it. See considerations on symmetric keys in Section 12.5.

The client instance's key MAY be pre-registered with the AS ahead of time and associated with a set of policies and allowable actions pertaining to that client. If this pre-registration includes other fields that can occur in the client request object described in this section, such as class_id or display, the pre-registered values MUST take precedence over any values given at runtime. Additional fields sent during a request but not present in a pre-registered client instance record at the AS SHOULD NOT be added to the client's pre-registered record. See additional considerations regarding client instance impersonation in Section 12.13.

A client instance that is capable of talking to multiple AS's SHOULD use a different key for each AS to prevent a class of mix-up attacks as described in Section 12.28.

2.3.1. Identifying the Client Instance by Reference

If the client instance has an instance identifier that the AS can use to determine appropriate key information, the client instance can send this instance identifier as a direct reference value in lieu of the client object. The instance identifier MAY be assigned to a client instance at runtime through a grant response (Section 3.5) or MAY be obtained in another fashion, such as a static registration process at the AS.

```
"client": "client-541-ab"
```

When the AS receives a request with an instance identifier, the AS MUST ensure that the key used to sign the request (Section 7.3) is associated with the instance identifier.

If the AS does not recognize the instance identifier, the request MUST be rejected with an error.

If the client instance is identified in this manner, the registered key for the client instance MAY be a symmetric key known to the AS. See considerations on symmetric keys in Section 12.5.

2.3.2. Providing Displayable Client Instance Information

If the client instance has additional information to display to the RO during any interactions at the AS, it MAY send that information in the "display" field. This field is a JSON object that declares information to present to the RO during any interactive sequences.

name (string): Display name of the client software. RECOMMENDED.

uri (string): User-facing web page of the client software.
   OPTIONAL.

logo_uri (string) Display image to represent the client software.
   The logo MAY be passed by value by using a data: URI [RFC2397]
   referencing an image mediatype. OPTIONAL.

```
"display": {
    "name": "My Client Display Name",
    "uri": "https://example.net/client",
    "logo_uri": "data:image/png;base64,Eeww...="
}
```

Additional display fields are defined by a registry TBD (Section 11).

The AS SHOULD use these values during interaction with the RO. The values are for informational purposes only and MUST NOT be taken as authentic proof of the client instance's identity or source. The AS MAY restrict display values to specific client instances, as identified by their keys in Section 2.3. See additional considerations for displayed client information in Section 12.13.

2.3.3. Authenticating the Client Instance

If the presented key is known to the AS and is associated with a single instance of the client software, the process of presenting a key and proving possession of that key is sufficient to authenticate the client instance to the AS. The AS MAY associate policies with the client instance identified by this key, such as limiting which resources can be requested and which interaction methods can be used. For example, only specific client instances with certain known keys might be trusted with access tokens without the AS interacting directly with the RO as in Appendix D.3.

The presentation of a key allows the AS to strongly associate multiple successive requests from the same client instance with each other. This is true when the AS knows the key ahead of time and can use the key to authenticate the client instance, but also if the key is ephemeral and created just for this series of requests. As such the AS MAY allow for client instances to make requests with unknown keys. This pattern allows for ephemeral client instances, such as single-page applications, and client software with many individual long-lived instances, such as mobile applications, to generate key pairs per instance and use the keys within the protocol without having to go through a separate registration step. The AS MAY limit which capabilities are made available to client instances with unknown keys. For example, the AS could have a policy saying that only previously-registered client instances can request particular resources, or that all client instances with unknown keys have to be interactively approved by an RO.

2.4. Identifying the User

If the client instance knows the identity of the end user through one or more identifiers or assertions, the client instance MAY send that information to the AS in the "user" field. The client instance MAY pass this information by value or by reference.

sub_ids (array of objects): An array of subject identifiers for the end user, as defined by [I-D.ietf-secevent-subject-identifiers]. OPTIONAL.

assertions (array of objects) An array containing assertions as objects each containing the assertion format and the assertion value as the JSON string serialization of the assertion. OPTIONAL.

```
"user": {
  "sub_ids": [ {
    "format": "opaque",
    "id": "J2G8G8O4AZ"
  } ],
  "assertions": [ {
    "format": "id_token",
    "value": "eyj..."
  } ]
}
```

Subject identifiers are hints to the AS in determining the RO and MUST NOT be taken as declarative statements that a particular RO is present at the client instance and acting as the end user. Assertions SHOULD be validated by the AS. [[See issue #49 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/49)]]

If the identified end user does not match the RO present at the AS during an interaction step, the AS SHOULD reject the request with an error.

[[See issue #50 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/50)]]

If the AS trusts the client instance to present verifiable assertions, the AS MAY decide, based on its policy, to skip interaction with the RO, even if the client instance provides one or more interaction modes in its request.

See Section 12.27 for considerations that the AS has to make when accepting and processing assertions from the client instance.

2.4.1. Identifying the User by Reference

The AS can identify the current end user to the client instance with a reference which can be used by the client instance to refer to the end user across multiple requests. If the client instance has a reference for the end user at this AS, the client instance MAY pass that reference as a string. The format of this string is opaque to the client instance.

```
"user": "XUT2MFM1XBIKJKSDU8QM"
```

One means of dynamically obtaining such a user reference is from the AS returning an opaque subject identifier as described in Section 3.4. Other means of configuring a client instance with a user identifier are out of scope of this specification.

User reference identifiers are not intended to be human-readable user identifiers or structured assertions. For the client instance to send either of these, use the full user request object (Section 2.4) instead.

If the AS does not recognize the user reference, it MUST return an error.

2.5. Interacting with the User

Often, the AS will require interaction with the RO (Section 4) in order to approve a requested delegation to the client instance for both access to resources and direct subject information. Many times the end user using the client instance is the same person as the RO, and the client instance can directly drive interaction with the end user by facilitating the process through means such as redirection to a URI or launching an application. Other times, the client instance can provide information to start the RO's interaction on a secondary device, or the client instance will wait for the RO to approve the request asynchronously. The client instance could also be signaled that interaction has concluded through a callback mechanism.

The client instance declares the parameters for interaction methods that it can support using the interact field.

The interact field is a JSON object with three keys whose values declare how the client can initiate and complete the request, as well as provide hints to the AS about user preferences such as locale. A client instance MUST NOT declare an interaction mode it does not support. The client instance MAY send multiple modes in the same request. There is no preference order specified in this request. An AS MAY respond to any, all, or none of the presented interaction modes (Section 3.3) in a request, depending on its capabilities and what is allowed to fulfill the request.

start (array of strings/objects): Indicates how the client instance can start an interaction. REQUIRED.

finish (object): Indicates how the client instance can receive an indication that interaction has finished at the AS. OPTIONAL.

hints (object): Provides additional information to inform the interaction process at the AS. OPTIONAL.

In this non-normative example, the client instance is indicating that it can redirect (Section 2.5.1.1) the end user to an arbitrary URI and can receive a redirect (Section 2.5.2.1) through a browser request.

```
"interact": {
    "start": ["redirect"],
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

In this non-normative example, the client instance is indicating that it can display a user code (Section 2.5.1.3) and direct the end user to an arbitrary URI (Section 2.5.1.1) on a secondary device, but it cannot accept a redirect or push callback.

```
"interact": {
    "start": ["redirect", "user_code"]
}
```

If the client instance does not provide a suitable interaction mechanism, the AS cannot contact the RO asynchronously, and the AS determines that interaction is required, then the AS SHOULD return an error since the client instance will be unable to complete the request without authorization.

The AS SHOULD handle any interact request as a one-time-use mechanism and SHOULD apply suitable timeouts to any interaction mechanisms provided, including user codes and redirection URIs. The client instance SHOULD apply suitable timeouts to any callback URIs.

2.5.1. Start Mode Definitions

This specification defines the following interaction start modes as an array of string values under the start key:

"redirect": Indicates that the client instance can direct the end user to an arbitrary URI for interaction. Section 2.5.1.1

"app": Indicates that the client instance can launch an application on the end user's device for interaction. Section 2.5.1.2

"user_code": Indicates that the client instance can communicate a human-readable short code to the end user for use with a stable URI. Section 2.5.1.3

"user_code_uri": Indicates that the client instance can communicate a human-readable short code to the end user for use with a short, dynamic URI. Section 2.5.1.4

2.5.1.1. Redirect to an Arbitrary URI

If the client instance is capable of directing the end user to a URI defined by the AS at runtime, the client instance indicates this by including redirect in the array under the start key. The means by which the client instance will activate this URI is out of scope of this specification, but common methods include an HTTP redirect, launching a browser on the end user's device, providing a scannable image encoding, and printing out a URI to an interactive console. While this URI is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the AS grant URI.

```
"interact": {
  "start": ["redirect"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a redirect interaction response Section 3.3.1. The client instance manages this interaction method as described in Section 4.1.1.

See Section 12.26 for more considerations regarding the use of front-channel communication techniques such as this.

2.5.1.2. Open an Application-specific URI

If the client instance can open a URI associated with an application on the end user's device, the client instance indicates this by including app in the array under the start key. The means by which the client instance determines the application to open with this URI are out of scope of this specification.

```
"interact": {
  "start": ["app"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns an app interaction response with an app URI payload Section 3.3.2. The client instance manages this interaction method as described in Section 4.1.4.

[[See issue #54 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/54)]]

2.5.1.3. Display a Short User Code

If the client instance is capable of displaying or otherwise communicating a short, human-entered code to the RO, the client instance indicates this by including user_code in the array under the start key. This code is to be entered at a static URI that does not change at runtime. The client instance has no reasonable means to communicate a dynamic URI to the RO, and so this URI is usually communicated out of band to the RO through documentation or other messaging outside of GNAP. While this URI is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the AS grant URI.

```
"interact": {
    "start": ["user_code"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a user code as specified in Section 3.3.3. The client instance manages this interaction method as described in Section 4.1.2.

2.5.1.4. Display a Short User Code and URI

If the client instance is capable of displaying or otherwise communicating a short, human-entered code along with a short, human-entered URI to the RO, the client instance indicates this by including user_code_uri in the array under the start key. This code is to be entered at the dynamic URL given in the response. While this URL is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the AS grant URL.

```
"interact": {
    "start": ["user_code_uri"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a user code and interaction URL as specified in Section 3.3.4. The client instance manages this interaction method as described in Section 4.1.3.

2.5.2. Finish Interaction Methods

If the client instance is capable of receiving a message from the AS indicating that the RO has completed their interaction, the client instance indicates this by sending the following members of an object under the finish key.

method (string): The callback method that the AS will use to contact the client instance. REQUIRED.

uri (string): Indicates the URI that the AS will either send the RO to after interaction or send an HTTP POST request. This URI MAY be unique per request and MUST be hosted by or accessible by the client instance. This URI MUST NOT contain any fragment component. This URI MUST be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme. If the client instance needs any state information to tie to the front channel interaction response, it MUST use a unique callback URI to link to that ongoing state. The allowable URIs and URI patterns MAY be restricted by the AS based on the client instance's presented key information. The callback URI SHOULD be presented to the RO during the interaction phase before redirect. REQUIRED for redirect and push methods.

nonce (string): Unique value to be used in the calculation of the "hash" query parameter sent to the callback URI, must be sufficiently random to be unguessable by an attacker. MUST be generated by the client instance as a unique value for this request. REQUIRED.

hash_method (string): The hash calculation mechanism to be used for the callback hash in Section 4.2.3. Can be one of sha3 or sha2. If absent, the default value is sha3. OPTIONAL. [[See issue #56 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/56)]]

This specification defines the following values for the method parameter, with other values defined by a registry TBD (Section 11):

"redirect": Indicates that the client instance can receive a redirect from the end user's device after interaction with the RO has concluded. Section 2.5.2.1

"push": Indicates that the client instance can receive an HTTP POST request from the AS after interaction with the RO has concluded. Section 2.5.2.2

If this interaction mode is supported for this client instance and request, the AS returns a nonce for use in validating the callback response (Section 3.3.5). Requests to the callback URI MUST be processed as described in Section 4.2, and the AS MUST require presentation of an interaction callback reference as described in Section 5.1.

[[See issue #58 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/58)]]

2.5.2.1. Receive an HTTP Callback Through the Browser

A finish method value of redirect indicates that the client instance will expect a request from the RO's browser using the HTTP method GET as described in Section 4.2.1.

```
"interact": {
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI MUST be processed by the client instance as described in Section 4.2.1.

Since the incoming request to the callback URI is from the RO's browser, this method is usually used when the RO and end user are the same entity. See Section 12.22 for considerations on ensuring the incoming HTTP message matches the expected context of the request. See Section 12.26 for more considerations regarding the use of front-channel communication techniques such as this.

2.5.2.2. Receive an HTTP Direct Callback

A finish method value of push indicates that the client instance will expect a request from the AS directly using the HTTP method POST as described in Section 4.2.2.

```
"interact": {
    "finish": {
        "method": "push",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI MUST be processed by the client instance as described in Section 4.2.2.

Since the incoming request to the callback URI is from the AS and not from the RO's browser, this request is not expected to have any shared session information from the start method. See Section 12.22 and Section 12.21 for more considerations regarding the use of back-channel and polling mechanisms like this.

2.5.3. Hints

The hints key is an object describing one or more suggestions from the client instance that the AS can use to help drive user interaction.

This specification defines the following properties under the hints key:

ui_locales (array of strings): Indicates the end user's preferred locales that the AS can use during interaction, particularly before the RO has authenticated. OPTIONAL. Section 2.5.3.1

The following sections detail requests for interaction hints. Additional interaction hints are defined in a registry TBD (Section 11).

2.5.3.1. Indicate Desired Interaction Locales

If the client instance knows the end user's locale and language preferences, the client instance can send this information to the AS using the ui_locales field with an array of locale strings as defined by [RFC5646].

```
"interact": {
    "hints": {
        "ui_locales": ["en-US", "fr-CA"]
    }
}
```

If possible, the AS SHOULD use one of the locales in the array, with preference to the first item in the array supported by the AS. If none of the given locales are supported, the AS MAY use a default locale.

2.5.4. Extending Interaction Modes

Additional interaction start modes, finish modes, and hints are defined in a registry TBD (Section 11).

2.6. Extending The Grant Request

The request object MAY be extended by registering new items in a registry TBD (Section 11). Extensions SHOULD be orthogonal to other parameters. Extensions MUST document any aspects where the extension item affects or influences the values or behavior of other request and response objects.

3. Grant Response

In response to a client instance's request, the AS responds with a JSON object as the HTTP entity body. Each possible field is detailed in the sections below.

continue (object): Indicates that the client instance can continue the request by making one or more continuation requests. REQUIRED if continuation calls are allowed for this client instance on this grant request. See Section 3.1.

access_token (object / array of objects): A single access token or set of access tokens that the client instance can use to call the RS on behalf of the RO. REQUIRED if an access token is included. See Section 3.2.

interact (object): Indicates that interaction through some set of defined mechanisms needs to take place. REQUIRED if interaction is needed or allowed. See Section 3.3.

subject (object): Claims about the RO as known and declared by the AS. REQUIRED if subject information is included. See Section 3.4.

instance_id (string): An identifier this client instance can use to identify itself when making future requests. OPTIONAL. See Section 3.5.

error (object): An error code indicating that something has gone wrong. REQUIRED for an error condition. If included, other fields MUST NOT be included. See Section 3.6.

In this example, the AS is returning an interaction URI (Section 3.3.1), a callback nonce (Section 3.3.5), and a continuation response (Section 3.1).

NOTE: '\' line wrapping per RFC 8792

```
   {
       "interact": {
           "redirect": "https://server.example.com/interact/4CF492ML\
             VMSW9MKMXKHQ",
           "finish": "MBDOFXG4Y5CVJCX821LH"
       },
       "continue": {
           "access_token": {
               "value": "80UPRY5NM33OMUKMKSKU",
           },
           "uri": "https://server.example.com/tx"
       }
   }
```

In this example, the AS is returning a bearer access token (Section 3.2.1) with a management URI and a subject identifier (Section 3.4) in the form of an opaque identifier.

NOTE: '\' line wrapping per RFC 8792 and a [subject identifier](#response-subje
an opaque identifier.

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "flags": ["bearer"],
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

In this example, the AS is returning set of subject identifiers (Section 3.4), simultaneously as an opaque identifier, an email address, and a decentralized identifier URL (DID).

```
{
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        }, {
          "format": "email",
          "email": "user@example.com"
        }, {
          "format": "did",
          "url": "did:example:123456"
        } ]
    }
}
```

3.1. Request Continuation

If the AS determines that the request can be continued with additional requests, it responds with the continue field. This field contains a JSON object with the following properties.

uri (string): The URI at which the client instance can make continuation requests. This URI MAY vary per request, or MAY be stable at the AS. The client instance MUST use this value exactly as given when making a continuation request (Section 5). REQUIRED.

wait (integer): The amount of time in integer seconds the client instance MUST wait after receiving this request continuation response and calling the continuation URI. The value SHOULD NOT be less than five seconds, and omission of the value MUST NOT be interpreted as zero (i.e., no delay between requests). RECOMMENDED.

access_token (object): A unique access token for continuing the request, called the "continuation access token". The value of this property MUST be in the format specified in Section 3.2.1. This access token MUST be bound to the client instance's key used in the request and MUST NOT be a bearer token. As a consequence, the flags array of this access token MUST NOT contain the string bearer and the key field MUST be omitted. The client instance MUST present the continuation access token in all requests to the continuation URI as described in Section 7.2. REQUIRED.

```
   {
       "continue": {
           "access_token": {
               "value": "80UPRY5NM33OMUKMKSKU"
           },
           "uri": "https://server.example.com/continue",
           "wait": 60
       }
   }
```

The client instance can use the values of this field to continue the request as described in Section 5. Note that the client instance MUST sign all continuation requests with its key as described in Section 7.3 and MUST present the access token in its continuation request.

This field SHOULD be returned when interaction is expected, to allow the client instance to follow up after interaction has been concluded.

3.2. Access Tokens

If the AS has successfully granted one or more access tokens to the client instance, the AS responds with the access_token field. This field contains either a single access token as described in Section 3.2.1 or an array of access tokens as described in Section 3.2.2.

The client instance uses any access tokens in this response to call the RS as described in Section 7.2.

3.2.1. Single Access Token

If the client instance has requested a single access token and the AS has granted that access token, the AS responds with the "access_token" field. The value of this field is an object with the following properties.

value (string): The value of the access token as a string. The value is opaque to the client instance. The value SHOULD be limited to ASCII characters to facilitate transmission over HTTP headers within other protocols without requiring additional encoding. REQUIRED.

label (string): The value of the label the client instance provided in the associated token request (Section 2.1), if present. If the token has been split by the AS, the value of the label field is chosen by the AS and the split flag is used. REQUIRED for multiple access tokens, OPTIONAL for single access token.

manage (string): The management URI for this access token. If provided, the client instance MAY manage its access token as described in Section 6. This management URI is a function of the AS and is separate from the RS the client instance is requesting access to. This URI MUST NOT include the access token value and SHOULD be different for each access token issued in a request. OPTIONAL.

access (array of objects/strings): A description of the rights associated with this access token, as defined in Section 8. If included, this MUST reflect the rights associated with the issued access token. These rights MAY vary from what was requested by the client instance. REQUIRED.

expires_in (integer): The number of seconds in which the access will expire. The client instance MUST NOT use the access token past this time. An RS MUST NOT accept an access token past this time. Note that the access token MAY be revoked by the AS or RS at any point prior to its expiration. OPTIONAL.

key (object / string): The key that the token is bound to, if different from the client instance's presented key. The key MUST be an object or string in a format described in Section 7.1. The client instance MUST be able to dereference or process the key information in order to be able to sign the request. OPTIONAL.

flags (array of strings): A set of flags that represent attributes or behaviors of the access token issued by the AS. OPTIONAL.

The values of the flags field defined by this specification are as follows:

"bearer": This flag indicates whether the token is a bearer token, not bound to a key and proofing mechanism. If the bearer flag is present, the access token is a bearer token, and the key field in this response MUST be omitted. If the bearer flag is omitted and the key field in this response is omitted, the token is bound the key used by the client instance (Section 2.3) in its request for access. If the bearer flag is omitted, and the key field is present, the token is bound to the key and proofing mechanism indicated in the key field. See Section 12.7 for additional considerations on the use of bearer tokens.

"durable": Flag indicating a hint of AS behavior on token rotation. If this flag is present, then the client instance can expect a previously-issued access token to continue to work after it has been rotated (Section 6.1) or the underlying grant request has been modified (Section 5.3), resulting in the issuance of new access tokens. If this flag is omitted, the client instance can anticipate a given access token could stop working after token rotation or grant request modification. Note that a token flagged as durable can still expire or be revoked through any normal means.

"split": Flag indicating that this token was generated by issuing multiple access tokens in response to one of the client instance's token request (Section 2.1) objects. This behavior MUST NOT be used unless the client instance has specifically requested it by use of the split flag.

Flag values MUST NOT be included more than once.

Additional flags can be defined by extensions using a registry TBD (Section 11).

The following non-normative example shows a single access token bound to the client instance's key used in the initial request, with a management URI, and that has access to three described resources (one using an object and two described by reference strings).

```
NOTE: '\' line wrapping per RFC 8792

"access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "manage": "https://server.example.com/token/PRY5NM33O\
        M4TB8N6BW7OZB8CDFONP219RP1L",
    "access": [
        {
            "type": "photo-api",
            "actions": [
                "read",
                "write",
                "dolphin"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        },
        "read", "dolphin-metadata"
    ]
}
```

The following non-normative example shows a single bearer access token with access to two described resources.

```
"access_token": {
    "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
    "flags": ["bearer"],
    "access": [
        "finance", "medical"
    ]
}
```

If the client instance requested a single access token (Section 2.1.1), the AS MUST NOT respond with the multiple access token structure unless the client instance sends the split flag as described in Section 2.1.1.

If the AS has split the access token response, the response MUST include the split flag.

[[See issue #69 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/69)]]

3.2.2. Multiple Access Tokens

If the client instance has requested multiple access tokens and the AS has granted at least one of them, the AS responds with the "access_token" field. The value of this field is a JSON array, the members of which are distinct access tokens as described in Section 3.2.1. Each object MUST have a unique label field, corresponding to the token labels chosen by the client instance in the multiple access token request (Section 2.1.2).

In this non-normative example, two tokens are issued under the names token1 and token2, and only the first token has a management URI associated with it.

NOTE: '\' line wrapping per RFC 8792

```
"access_token": [
    {
        "label": "token1",
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [ "finance" ]
    },
    {
        "label": "token2",
        "value": "UFGLO2FDAFG7VGZZPJ3IZEMN21EVU71FHCARP4J1",
        "access": [ "medical" ]
    }
}
```

Each access token corresponds to one of the objects in the access_token array of the client instance's request (Section 2.1.2).

The multiple access token response MUST be used when multiple access tokens are requested, even if only one access token is issued as a result of the request. The AS MAY refuse to issue one or more of the requested access tokens, for any reason. In such cases the refused token is omitted from the response and all of the other issued access tokens are included in the response the requested names appropriate names.

If the client instance requested multiple access tokens (Section 2.1.2), the AS MUST NOT respond with a single access token structure, even if only a single access token is granted. In such cases, the AS responds with a multiple access token structure containing one access token.

If the AS has split the access token response, the response MUST include the split flag in the flags array.

```
"access_token": [
    {
        "label": "split-1",
        "value": "8N6BW7OZB8CDFONP219-OS9M2PMHKUR64TBRP1LT0",
        "flags": ["split"],
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [ "fruits" ]
    },
    {
        "label": "split-2",
        "value": "FG7VGZZPJ3IZEMN21EVU71FHCAR-UFGLO2FDAP4J1",
        "flags": ["split"],
        "access": [ "vegetables" ]
    }
}
```

Each access token MAY be bound to different keys with different proofing mechanisms.

The manage URI MUST NOT contain the access token value.

3.3. Interaction Modes

If the client instance has indicated a capability to interact with the RO in its request (Section 2.5), and the AS has determined that interaction is both supported and necessary, the AS responds to the client instance with any of the following values in the interact field of the response. There is no preference order for interaction modes in the response, and it is up to the client instance to determine which ones to use. All supported interaction methods are included in the same interact object.

redirect (string): Redirect to an arbitrary URI. REQUIRED if the redirect interaction start mode is possible for this request. See Section 3.3.1.

app (string): Launch of an application URI. REQUIRED if the app interaction start mode is possible for this request. See Section 3.3.2.

user_code (object): Display a short user code. REQUIRED if the user_code interaction start mode is possible for this request. See Section 3.3.3.

user_code_uri (object): Display a short user code and URL. REQUIRED if the user_code_uri interaction start mode is possible for this request. Section 3.3.4

finish (string): A nonce used by the client instance to verify the callback after interaction is completed. REQUIRED if the interaction finish method requested by the client instance is possible for this request. See Section 3.3.5.

Additional interaction mode responses can be defined in a registry TBD (Section 11).

The AS MUST NOT respond with any interaction mode that the client instance did not indicate in its request. The AS MUST NOT respond with any interaction mode that the AS does not support. Since interaction responses include secret or unique information, the AS SHOULD respond to each interaction mode only once in an ongoing request, particularly if the client instance modifies its request (Section 5.3).

3.3.1. Redirection to an arbitrary URI

If the client instance indicates that it can redirect to an arbitrary URI (Section 2.5.1.1) and the AS supports this mode for the client instance's request, the AS responds with the "redirect" field, which is a string containing the URI to direct the end user to. This URI MUST be unique for the request and MUST NOT contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "redirect": "https://interact.example.com/4CF492MLVMSW9MKMXKHQ"
}
```

The URI returned is a function of the AS, but the URI itself MAY be completely distinct from the URI the client instance uses to request access (Section 2), allowing an AS to separate its user-interactive functionality from its back-end security functionality. If the AS does not directly host the functionality accessed through the given URI, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

[[See issue #72 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/72)]]

The client instance sends the end user to the URI to interact with the AS. The client instance MUST NOT alter the URI in any way. The means for the client instance to send the end user to this URI is out of scope of this specification, but common methods include an HTTP redirect, launching the system browser, displaying a scannable code, or printing out the URI in an interactive console. See details of the interaction in Section 4.1.1.

3.3.2. Launch of an application URI

If the client instance indicates that it can launch an application URI (Section 2.5.1.2) and the AS supports this mode for the client instance's request, the AS responds with the "app" field, which is a string containing the URI for the client instance to launch. This URI MUST be unique for the request and MUST NOT contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "app": "https://app.example.com/launch?tx=4CF492MLV"
}
```

The means for the launched application to communicate with the AS are out of scope for this specification.

The client instance launches the URI as appropriate on its platform, and the means for the client instance to launch this URI is out of scope of this specification. The client instance MUST NOT alter the URI in any way. The client instance MAY attempt to detect if an installed application will service the URI being sent before attempting to launch the application URI. See details of the interaction in Section 4.1.4.

[[See issue #71 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/71)]]

3.3.3. Display of a Short User Code

If the client instance indicates that it can display a short user-typeable code (Section 2.5.1.3) and the AS supports this mode for the client instance's request, the AS responds with a "user_code" field. This field is an object that contains the following members.

code (string): A unique short code that the user can type into a web page. This string MUST be case-insensitive, MUST consist of only easily typeable characters (such as letters or numbers). The time in which this code will be accepted SHOULD be short lived, such as several minutes. It is RECOMMENDED that this code be no more than eight characters in length. REQUIRED.

```
"interact": {
    "user_code": {
        "code": "A1BC-3DFF",
    }
}
```

The client instance MUST communicate the "code" to the end user in some fashion, such as displaying it on a screen or reading it out audibly. This code is used by the interaction component of the AS as a means of identifying the pending grant request and does not function as an authentication factor for the RO.

The URI that the end user is intended to enter the code into MUST be stable, since the client instance is expected to have no means of communicating a dynamic URI to the end user at runtime.

As this interaction mode is designed to facilitate interaction via a secondary device, it is not expected that the client instance redirect the end user to the URL given here at runtime. If the client instance is capable of communicating an short arbitrary URI to the end user for use with the user code, the client instance can instead use the "user_code_uri" (Section 2.5.1.4) method instead. If the client instance is capable of communicating a long arbitrary URI to the end user, such as through a scannable code, the client instance can use the "redirect" (Section 2.5.1.1) mode for this purpose instead of or in addition to the user code mode.

See details of the interaction in Section 4.1.2.

3.3.4. Display of a Short User Code and URI

If the client instance indicates that it can display a short user-typeable code (Section 2.5.1.3) and the AS supports this mode for the client instance's request, the AS responds with a "user_code_uri" object that contains the following members.

code (string): A unique short code that the end user can type into a provided URI. This string MUST be case-insensitive, MUST consist of only easily typeable characters (such as letters or numbers). The time in which this code will be accepted SHOULD be short lived, such as several minutes. It is RECOMMENDED that this code be no more than eight characters in length. REQUIRED.

uri (string): The interaction URI that the client instance will direct the RO to. This URI MUST be short enough to be communicated to the end user. It is RECOMMENDED that this URI be short enough for an end user to type in manually. The URI MUST NOT contain the code value. REQUIRED.

```
"interact": {
    "user_code_uri": {
        "code": "A1BC-3DFF",
        "uri": "https://srv.ex/device"
    }
}
```

The client instance MUST communicate the "code" to the end user in some fashion, such as displaying it on a screen or reading it out audibly. This code is used by the interaction component of the AS as a means of identifying the pending grant request and does not function as an authentication factor for the RO.

The client instance MUST also communicate the URI to the end user. Since it is expected that the end user will continue interaction on a secondary device, the URI needs to be short enough to allow the end user to type or copy it to a secondary device without mistakes.

The URI returned is a function of the AS, but the URI itself MAY be completely distinct from the URI the client instance uses to request access (Section 2), allowing an AS to separate its user-interactive functionality from its back-end security functionality. If the AS does not directly host the functionality accessed through the given URI, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

See details of the interaction in Section 4.1.2.

3.3.5. Interaction Finish

If the client instance indicates that it can receive a post-interaction redirect or push at a URI (Section 2.5.2) and the AS supports this mode for the client instance's request, the AS responds with a finish field containing a nonce that the client instance will use in validating the callback as defined in Section 4.2.

```
"interact": {
    "finish": "MBDOFXG4Y5CVJCX821LH"
}
```

When the interaction is completed, the interaction component MUST contact the client instance using either a redirect or launch of the RO's browser or through an HTTP POST to the client instance's callback URI using the method indicated in the interaction request (Section 2.5.2) as described in Section 4.2.

If the AS returns a nonce, the client instance MUST NOT continue a grant request before it receives the associated interaction reference on the callback URI. See details in Section 4.2.

3.3.6. Extending Interaction Mode Responses

Extensions to this specification can define new interaction mode responses in a registry TBD (Section 11). Extensions MUST document the corresponding interaction request.

3.4. Returning Subject Information

If information about the RO is requested and the AS grants the client instance access to that data, the AS returns the approved information in the "subject" response field. The AS MUST return the subject field only in cases where the AS is sure that the RO and the end user are the same party. This can be accomplished through some forms of interaction with the RO (Section 4).

This field is an object with the following OPTIONAL properties.

sub_ids (array of objects): An array of subject identifiers for the RO, as defined by [I-D.ietf-secevent-subject-identifiers]. REQUIRED if returning subject identifiers.

assertions (array of objects): An array containing assertions as objects each containing the assertion format and the assertion value as the JSON string serialization of the assertion. Possible formats include id_token for an [OIDC] ID Token and saml2 for a SAML 2 assertion. Additional assertion formats are defined by a registry TBD (Section 11). REQUIRED if returning assertions.

updated_at (string): Timestamp as an ISO8610 date string, indicating when the identified account was last updated. The client instance MAY use this value to determine if it needs to request updated profile information through an identity API. The definition of such an identity API is out of scope for this specification. RECOMMENDED.

```
"subject": {
  "sub_ids": [ {
    "format": "opaque",
    "id": "XUT2MFM1XBIKJKSDU8QM"
  } ],
  "assertions": [ {
    "format": "id_token",
    "value": "eyj..."
  } ]
}
```

Subject identifiers returned by the AS SHOULD uniquely identify the RO at the AS. Some forms of subject identifier are opaque to the client instance (such as the subject of an issuer and subject pair), while others forms (such as email address and phone number) are intended to allow the client instance to correlate the identifier with other account information at the client instance. The AS MUST ensure that the returned subject identifiers only apply to the authenticated end user. The client instance MUST NOT request or use any returned subject identifiers for communication purposes (see Section 2.2). That is, a subject identifier returned in the format of an email address or a phone number only identifies the RO to the AS and does not indicate that the AS has validated that the represented email address or phone number in the identifier is suitable for communication with the current user. To get such information, the client instance MUST use an identity protocol to request and receive additional identity claims. The details of an identity protocol and associated schema are outside the scope of this specification.

Extensions to this specification MAY define additional response properties in a registry TBD (Section 11).

See Section 12.27 for considerations that the client instance has to make when accepting and processing assertions from the AS.

3.5. Returning a Dynamically-bound Client Instance Identifier

Many parts of the client instance's request can be passed as either a value or a reference. The use of a reference in place of a value allows for a client instance to optimize requests to the AS.

Some references, such as for the client instance's identity (Section 2.3.1) or the requested resources (Section 8.1), can be managed statically through an admin console or developer portal provided by the AS or RS. The developer of the client software can include these values in their code for a more efficient and compact request.

If desired, the AS MAY also generate and return an instance identifier dynamically to the client instance in the response to facilitate multiple interactions with the same client instance over time. The client instance SHOULD use this instance identifier in future requests in lieu of sending the associated data values in the client field.

Dynamically generated client instance identifiers are string values that MUST be protected by the client instance as secrets. Instance identifier values MUST be unguessable and MUST NOT contain any information that would compromise any party if revealed. Instance identifier values are opaque to the client instance.

instance_id (string): A string value used to represent the information in the client object that the client instance can use in a future request, as described in Section 2.3.1. OPTIONAL.

This non-normative example shows an instance identifier along side an issued access token.

```
{
    "instance_id": "7C7C4AZ9KHRS6X63AJAO",
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0"
    }
}
```

[[See issue #77 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/77)]]

[[See issue #78 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/78)]]

3.6. Error Response

If the AS determines that the request cannot be issued for any reason, it responds to the client instance with an error message.

error (string): A single ASCII error code from the following, with additional values available in a registry TBD (Section 11). REQUIRED.

"invalid_request": The request is missing a required parameter, includes an invalid parameter value or is otherwise malformed.

"invalid_client": The request was made from a client that was not recognized or allowed by the AS, or the client's signature validation failed.

"user_denied": The RO denied the request.

"too_fast": The client instance did not respect the timeout in the wait response.

"unknown_request": The request referenced an unknown ongoing access request.

"request_denied": The request was denied for an unspecified reason.

error_description (string): A human-readable string description of the error intended for the developer of the client. OPTIONAL.

For example, if the RO denied the request while interacting with the AS, the AS would return the following error when the client instance tries to continue the grant request:

```
{
  "error": "user_denied"
}
```

3.7. Extending the Response

Extensions to this specification MAY define additional fields for the grant response in a registry TBD (Section 11).

4. Determining Authorization and Consent

When the client instance makes its initial request (Section 2) to the AS for delegated access, it is capable of asking for several different kinds of information in response:

* the access being requested in the access_token request parameter

* the subject information being requested in the subject request parameter

* any additional requested information defined by extensions of this protocol

The AS determines what authorizations and consents are required to fulfill this requested delegation. The details of how the AS makes this determination are out of scope for this document. However, there are several common patterns defined and supported by GNAP for fulfilling these requirements, including information sent by the client instance, information gathered through the interaction process, and information supplied by external parties. An individual AS can define its own policies and processes for deciding when and how to gather the necessary authorizations and consent.

The client instance can supply information directly to the AS in its request. From this information, the AS can determine if the requested delegation can be granted immediately. The client instance can send several kinds of things, including:

* the identity of the client instance, known from the presented keys or associated identifiers

* the identity of the end user presented in the user request parameter

* any additional information presented by the client instance in the request, including any extensions

The AS will verify this presented information in the context of the client instance's request and can only trust the information as much as it trusts the presentation and context of the information. If the AS determines that the information presented in the initial request is sufficient for granting the requested access, the AS MAY return the positive results immediately in its response (Section 3) with access tokens and subject information.

If the AS determines that additional runtime authorization is required, the AS can either deny the request outright or use a number of means at its disposal to gather that authorization from the appropriate ROs, including for example:

* starting interaction with the end user facilitated by the client software, such as a redirection or user code

* challenging the client instance through a challenge-response mechanism

* requesting that the client instance present specific additional information, such as a user's credential or an assertion

* contacting an RO through an out-of-band mechanism, such as a push notification

* contacting an auxiliary software process through an out-of-band mechanism, such as querying a digital wallet

The authorization and consent gathering process in GNAP is left deliberately flexible to allow for a wide variety of different deployments, interactions, and methodologies. In this process, the AS can gather consent from the RO as necessitated by the access that has been requested. The AS can sometimes determine which RO needs to consent based on what has been requested by the client instance, such as a specific RS record, an identified user, or a request requiring specific access such as approval by an administrator. If the AS has a means of contacting the RO directly, it could do so without involving the client instance in its consent gathering process. For example, the AS could push a notification to a known RO and have the RO approve the pending request asynchronously. These interactions can be through an interface of the AS itself (such as a hosted web page), through another application (such as something installed on the RO's device), through a messaging fabric, or any other means. When interacting with an RO, the AS can do anything it needs to determine the authorization of the requested grant, including:

* authenticate the RO, through a local account or some other means such as federated login

* validate the RO through presentation of claims, attributes, or other information

* prompt the RO for consent for the requested delegation

* describe to the RO what information is being released, to whom, and for what purpose

* provide warnings to the RO about potential attacks or negative effects of allowing the information

* allow the RO to modify the client instance's requested access, including limiting or expanding that access

* provide the RO with artifacts such as receipts to facilitate an audit trail of authorizations

* allow the RO to deny the requested delegation

The AS is also allowed to request authorization from more than one RO, if the AS deems fit. For example, a medical record might need to be released by both an attending nurse and a physician, or both owners of a bank account need to sign off on a transfer request. Alternatively, the AS could require N of M possible RO's to approve a given request in order. The AS could also determine that the end user is not the appropriate RO for a given request and reach out to the appropriate RO asynchronously. The details of determining which RO's are required for a given request are out of scope for this specification.

The client instance can also indicate that it is capable of facilitating interaction with the end user, another party, or another piece of software through its interaction start (Section 2.5.1) request. In many cases, the end user is delegating their own access as RO to the client instance. Here, the AS needs to determine the identity of the end user and will often need to interact directly with the end user to determine their status as an RO and collect their consent. If the AS has determined that authorization is required and the AS can support one or more of the requested interaction start methods, the AS returns the associated interaction start responses (Section 3.3). The client instance SHOULD initiate one or more of these interaction methods (Section 4.1) in order to facilitate the granting of the request. If more than one interaction start method is available, the means by which the client chooses which methods to follow is out of scope of this specification. The client instance MUST use each interaction method once at most.

After starting interaction, the client instance can then make a continuation request (Section 5) either in response to a signal indicating the finish of the interaction (Section 4.2), through polling, or through some other method defined by an extension of this specification.

If the AS and client instance have not reached a state where the delegation can be granted, the AS and client instance can repeat the interaction process as long as the AS supplies the client instance with continuation information (Section 3.1) to facilitate the ongoing requests.

4.1. Interaction Start Methods

To initiate an interaction start method indicated by the interaction start responses (Section 3.3) from the AS, the client instance follows the steps defined by that interaction method. The actions of the client instance required for the interaction start modes defined in this specification are described in the following sections.

4.1.1. Interaction at a Redirected URI

When the end user is directed to an arbitrary URI through the "redirect" (Section 3.3.1) mode, the client instance facilitates opening the URI through the end user's web browser. The client instance could launch the URI through the system browser, provide a clickable link, redirect the user through HTTP response codes, or display the URI in a form the end user can use to launch such as a multidimensional barcode. With this method, it is common (though not required) for the RO to be the same party as the end user, since the client instance has to communicate the redirection URI to the end user.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The URI value is used to identify the grant request being authorized. If the URI cannot be associated with a currently active request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any client instance even if a redirect finish method is supplied (Section 2.5.2.1). If the URI is not hosted by the AS directly, the means of communication between the AS and this URI are out of scope for this specification.

The client instance MUST NOT modify the URI when launching it, in particular the client instance MUST NOT add any parameters to the URI. The URI MUST be reachable from the end user's browser, though the URI MAY be opened on a separate device from the client instance itself. The URI MUST be accessible from an HTTP GET request and MUST be protected by HTTPS or equivalent means.

4.1.2. Interaction at the Static User Code URI

When the end user is directed to enter a short code through the "user_code" (Section 3.3.3) mode, the client instance communicates the user code to the end user and directs the end user to enter that code at an associated URI. This mode is used when the client instance is not able to communicate or facilitate launching an arbitrary URI. The associated URI could be statically configured with the client instance or in the client software's documentation. As a consequence, these URIs SHOULD be short. The user code URI MUST be reachable from the end user's browser, though the URI is usually be opened on a separate device from the client instance itself. Since it is designed to be typed in, the URI SHOULD be accessible from an HTTP GET request and MUST be protected by HTTPS or equivalent means.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The value of the user code is used to identify the grant request being authorized. If the user code cannot be associated with a currently active request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any client instance even if a redirect finish method is supplied (Section 2.5.2.1). If the interaction component at the user code URI is not hosted by the AS directly, the means of communication between the AS and this URI, including communication of the user code itself, are out of scope for this specification.

When the RO enters this code at the user code URI, the AS MUST uniquely identify the pending request that the code was associated with. If the AS does not recognize the entered code, the interaction component MUST display an error to the user. If the AS detects too many unrecognized code enter attempts, the interaction component SHOULD display an error to the user and MAY take additional actions such as slowing down the input interactions. The user should be warned as such an error state is approached, if possible.

4.1.3. Interaction at a Dynamic User Code URI

When the end user is directed to enter a short code through the "user_code_uri" (Section 3.3.4) mode, the client instance communicates the user code and associated URI to the end user and directs the end user to enter that code at the URI. This mode is used when the client instance is not able to facilitate launching an arbitrary URI but can communicate arbitrary values like URIs. As a consequence, these URIs SHOULD be short. The client instance MUST NOT modify the URI when communicating it to the end user; in particular the client instance MUST NOT add any parameters to the URI. The user code URI MUST be reachable from the end user's browser, though the URI is usually be opened on a separate device from the client instance itself. Since it is designed to be typed in, the URI SHOULD be accessible from an HTTP GET request and MUST be protected by HTTPS or equivalent means.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. The value of the user code is used to identify the grant request being authorized. If the user code cannot be associated with a currently active request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any client instance even if a redirect finish method is supplied (Section 2.5.2.1). If the interaction component at the user code URI is not hosted by the AS directly, the means of communication between the AS and this URI, including communication of the user code itself, are out of scope for this specification.

When the RO enters this code at the given URI, the AS MUST uniquely identify the pending request that the code was associated with. If the AS does not recognize the entered code, the interaction component MUST display an error to the user. If the AS detects too many unrecognized code enter attempts, the interaction component SHOULD display an error to the user and MAY take additional actions such as slowing down the input interactions. The user should be warned as such an error state is approached, if possible.

4.1.4. Interaction through an Application URI

When the client instance is directed to launch an application through the "app" (Section 3.3.2) mode, the client launches the URI as appropriate to the system, such as through a deep link or custom URI scheme registered to a mobile application. The means by which the AS and the launched application communicate with each other and perform any of the required actions are out of scope for this specification.

4.2. Post-Interaction Completion

If an interaction "finish" (Section 3.3.5) method is associated with the current request, the AS MUST follow the appropriate method at upon completion of interaction in order to signal the client instance to continue, except for some limited error cases discussed below. If a finish method is not available, the AS SHOULD instruct the RO to return to the client instance upon completion.

The AS MUST create an interaction reference and associate that reference with the current interaction and the underlying pending request. This interaction reference value MUST be sufficiently random so as not to be guessable by an attacker. The interaction reference MUST be one-time-use to prevent interception and replay attacks.

The AS MUST calculate a hash value based on the client instance and AS nonces and the interaction reference, as described in Section 4.2.3. The client instance will use this value to validate the "finish" call.

The AS MUST send the hash and interaction reference based on the interaction finish mode as described in the following sections.

Note that the "finish" method still occurs in many error cases, such as when the RO has denied access. This pattern allows the client instance to potentially recover from the error state by modifying its request or providing additional information directly to the AS in a continuation request. The AS MUST NOT follow the "finish" method in the following circumstances:

* The AS has determined that any URIs involved with the finish method are dangerous or blocked.

* The AS cannot determine which ongoing grant request is being referenced.

* The ongoing grant request has been cancelled or otherwise blocked.

4.2.1. Completing Interaction with a Browser Redirect to the Callback URI

When using the redirect interaction finish method (Section 3.3.5), the AS signals to the client instance that interaction is complete and the request can be continued by directing the RO (in their browser) back to the client instance's redirect URI sent in the callback request (Section 2.5.2.1).

The AS secures this redirect by adding the hash and interaction reference as query parameters to the client instance's redirect URI.

hash: The interaction hash value as described in Section 4.2.3. REQUIRED.

interact_ref: The interaction reference generated for this interaction. REQUIRED.

The means of directing the RO to this URI are outside the scope of this specification, but common options include redirecting the RO from a web page and launching the system browser with the target URI. See Section 12.16 for considerations on which HTTP status code to use when redirecting a request that potentially contains credentials.

NOTE: '\' line wrapping per RFC 8792

```
https://client.example.net/return/123455\
  ?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2\
    HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

When receiving the request, the client instance MUST parse the query parameters to calculate and validate the hash value as described in Section 4.2.3. If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.2.2. Completing Interaction with a Direct HTTP Request Callback

When using the push interaction finish method (Section 3.3.5), the AS signals to the client instance that interaction is complete and the request can be continued by sending an HTTP POST request to the client instance's callback URI sent in the callback request (Section 2.5.2.2).

The entity message body is a JSON object consisting of the following two fields:

hash (string): The interaction hash value as described in Section 4.2.3. REQUIRED.

interact_ref (string) The interaction reference generated for this interaction. REQUIRED.

NOTE: '\' line wrapping per RFC 8792

```
POST /push/554321 HTTP/1.1
Host: client.example.net
Content-Type: application/json

{
  "hash": "p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R\
    2HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A",
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

When processing such a call, the AS MUST protect itself against SSRF attacks as discussed in Section 12.31.

When receiving the request, the client instance MUST parse the JSON object and validate the hash value as described in Section 4.2.3. If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.2.3. Calculating the interaction hash

The "hash" parameter in the request to the client instance's callback URI ties the front channel response to an ongoing request by using values known only to the parties involved. This security mechanism allows the client instance to protect itself against several kinds of session fixation and injection attacks. The AS MUST always provide this hash, and the client instance MUST validate the hash when received.

To calculate the "hash" value, the party doing the calculation creates a hash string by concatenating the following values in the following order using a single newline (\n) character to separate them:

* the "nonce" value sent by the client instance in the interaction "finish" section of the initial request (Section 2.5.2)

* the AS's nonce value from the interaction finish response (Section 3.3.5)

* the "interact_ref" returned from the AS as part of the interaction finish method (Section 4.2)

* the grant endpoint URI the client instance used to make its initial request (Section 2)

There is no padding or whitespace before or after any of the lines, and no trailing newline character.

```
VJLO6A4CAYLBXHTR0KRO
MBDOFXG4Y5CVJCX821LH
4IFWWIKYBC2PQ6U56NL1
https://server.example.com/tx
```

The party then hashes this string with the appropriate algorithm based on the "hash_method" parameter under the "finish" key. If the "hash_method" value is not present in the client instance's request, the algorithm defaults to "sha3".

[[See issue #56 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/56)]]

4.2.3.1. SHA3-512

The "sha3" hash method consists of hashing the input string with the 512-bit SHA3 algorithm. The byte array is then encoded using URL Safe Base64 with no padding [RFC4648]. The resulting string is the hash value.

NOTE: '\' line wrapping per RFC 8792

```
p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2HZT8BOWYHcLmObM\
  7XHPAdJzTZMtKBsaraJ64A
```

4.2.3.2. SHA2-512

The "sha2" hash method consists of hashing the input string with the 512-bit SHA2 algorithm. The byte array is then encoded using URL Safe Base64 with no padding [RFC4648]. The resulting string is the hash value.

NOTE: '\' line wrapping per RFC 8792

```
62SbcD3Xs7L40rjgALA-ymQujoh2LB2hPJyX9vlcr1H6ecChZ8BNKkG_HrOKP_Bp\
  j84rh4mC9aE9x7HPBFcIHw
```

5. Continuing a Grant Request

While it is possible for the AS to return a grant response (Section 3) with all the client instance's requested information (including access tokens (Section 3.2) and direct user information (Section 3.4)), it's more common that the AS and the client instance will need to communicate several times over the lifetime of an access grant. This is often part of facilitating interaction (Section 4), but it could also be used to allow the AS and client instance to continue negotiating the parameters of the original grant request (Section 2).

To enable this ongoing negotiation, the AS provides a continuation API to the client software. The AS returns a continue field in the response (Section 3.1) that contains information the client instance needs to access this API, including a URI to access as well as a continuation access token to use during the requests.

The continuation access token is initially bound to the same key and method the client instance used to make the initial request. As a consequence, when the client instance makes any calls to the continuation URI, the client instance MUST present the continuation access token as described in Section 7.2 and present proof of the client instance's key (or its most recent rotation) by signing the request as described in Section 7.3. The AS MUST validate all keys presented by the client instance or referenced in an ongoing request for each call within that request.

Access tokens other than the continuation access tokens MUST NOT be usable for continuation requests.

[[See issue #85 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/85)]]

For example, here the client instance makes a POST request to a unique URI and signs the request with HTTP Message Signatures:

```
POST /continue/KSKUOMUKM HTTP/1.1
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Host: server.example.com
Signature-Input: sig1=...
Signature: sig1=...
```

The AS MUST be able to tell from the client instance's request which specific ongoing request is being accessed, using a combination of the continuation URI, the provided continuation access token, and the client instance identified by the key signature. If the AS cannot determine a single active grant request to map the continuation request to, the AS MUST return an error.

The ability to continue an already-started request allows the client instance to perform several important functions, including presenting additional information from interaction, modifying the initial request, and getting the current state of the request.

All requests to the continuation API are protected by this bound continuation access token. For example, here the client instance makes a POST request to a stable continuation endpoint URI with the interaction reference (Section 5.1), includes the access token, and signs with HTTP Message Signatures:

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

If a wait parameter was included in the continuation response (Section 3.1), the client instance MUST NOT call the continuation URI prior to waiting the number of seconds indicated. If no wait period is indicated, the client instance MUST NOT poll immediately and SHOULD wait at least 5 seconds. If the client instance does not respect the given wait period, the AS MUST return the error too_fast defined in Section 3.6.

The response from the AS is a JSON object and MAY contain any of the fields described in Section 3, as described in more detail in the sections below.

If the AS determines that the client instance can make a further continuation request, the AS MUST include a new "continue" response (Section 3.1). The new continue response MUST include a continuation access token as well, and this token SHOULD be a new access token, invalidating the previous access token. If the AS does not return a new continue response, the client instance MUST NOT make an additional continuation request. If a client instance does so, the AS MUST return an error. [[See issue #87 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/87)]]

For continuation functions that require the client instance to send a message body, the body MUST be a JSON object.

5.1. Continuing After a Completed Interaction

When the AS responds to the client instance's finish method as in Section 4.2.1, this response includes an interaction reference. The client instance MUST include that value as the field interact_ref in a POST request to the continuation URI.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

Since the interaction reference is a one-time-use value as described in Section 4.2.1, if the client instance needs to make additional continuation calls after this request, the client instance MUST NOT include the interaction reference. If the AS detects a client instance submitting the same interaction reference multiple times, the AS MUST return an error and SHOULD invalidate the ongoing request.

The grant response (Section 3) MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. The response SHOULD NOT contain any interaction responses (Section 3.3). [[See issue #89 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/89)]]

For example, if the request is successful in causing the AS to issue access tokens and release opaque subject claims, the response could look like this:

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

With this example, the client instance can not make an additional continuation request because a continue field is not included.

[[See issue #88 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/88)]]

5.2. Continuing During Pending Interaction

When the client instance does not include a finish parameter, the client instance will often need to poll the AS until the RO has authorized the request. To do so, the client instance makes a POST request to the continuation URI as in Section 5.1, but does not include a message body.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

The grant response (Section 3) MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If a continue field is included, it SHOULD include a wait field to facilitate a reasonable polling rate by the client instance. The response SHOULD NOT contain interaction responses (Section 3.3).

For example, if the request has not yet been authorized by the RO, the AS could respond by telling the client instance to make another continuation request in the future. In this example, a new, unique access token has been issued for the call, which the client instance will use in its next continuation request.

```
{
    "continue": {
        "access_token": {
            "value": "33OMUKMKSKU80UPRY5NM"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    }
}
```

[[See issue #90 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/90)]]

[[See issue #91 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/91)]]

If the request is successful in causing the AS to issue access tokens and release subject claims, the response could look like this example:

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
          "format": "opaque",
          "id": "J2G8G8O4AZ"
        } ]
    }
}
```

See Section 12.21 for considerations on polling for continuation without an interaction finish method.

5.3. Modifying an Existing Request

The client instance might need to modify an ongoing request, whether or not tokens have already been issued or claims have already been released. In such cases, the client instance makes an HTTP PATCH request to the continuation URI and includes any fields it needs to modify. Fields that aren't included in the request are considered unchanged from the original request.

The client instance MAY include the access_token and subject fields as described in Section 2.1 and Section 2.2. Inclusion of these fields override any values in the initial request, which MAY trigger additional requirements and policies by the AS. For example, if the client instance is asking for more access, the AS could require additional interaction with the RO to gather additional consent. If the client instance is asking for more limited access, the AS could determine that sufficient authorization has been granted to the client instance and return the more limited access rights immediately. [[See issue #92 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/92)]]

The client instance MAY include the interact field as described in Section 2.5. Inclusion of this field indicates that the client instance is capable of driving interaction with the RO, and this field replaces any values from a previous request. The AS MAY respond to any of the interaction responses as described in Section 3.3, just like it would to a new request.

The client instance MAY include the user field as described in Section 2.4 to present new assertions or information about the end user. [[See issue #93 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/93)]]

The client instance MUST NOT include the client section of the request. [[See issue #94 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/94)]]

The client instance MAY include post-interaction responses such as described in Section 5.1. [[See issue #95 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/95)]]

Modification requests MUST NOT alter previously-issued access tokens. Instead, any access tokens issued from a continuation are considered new, separate access tokens. The AS MAY revoke existing access tokens after a modification has occurred. [[See issue #96 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/96)]]

If the modified request can be granted immediately by the AS, the grant response (Section 3) MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If interaction can occur, the response SHOULD contain interaction responses (Section 3.3) as well.

For example, a client instance initially requests a set of resources using references:

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a continue field, which includes a separate access token for accessing the continuation API:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read", "write"
        ]
    }
}
```

This continue field allows the client instance to make an eventual continuation call. In the future, the client instance realizes that it no longer needs "write" access and therefore modifies its ongoing request, here asking for just "read" access instead of both "read" and "write" as before.

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    }
    ...
}
```

The AS replaces the previous access from the first request, allowing the AS to determine if any previously-granted consent already applies. In this case, the AS would likely determine that reducing the breadth of the requested access means that new access tokens can be issued to the client instance. The AS would likely revoke previously-issued access tokens that had the greater access rights associated with them, unless they had been issued with the durable flag.

```
{
    "continue": {
        "access_token": {
            "value": "M33OMUK80UPRY5NMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "0EVKC7-2ZKwZM_6N760",
        "access": [
            "read"
        ]
    }
}
```

For another example, the client instance initially requests read-only access but later needs to step up its access. The initial request could look like this example.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a continue field:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read"
        ]
    }
}
```

This allows the client instance to make an eventual continuation call. The client instance later realizes that it now needs "write" access in addition to the "read" access. Since this is an expansion of what it asked for previously, the client instance also includes a new interaction section in case the AS needs to interact with the RO again to gather additional authorization. Note that the client instance's nonce and callback are different from the initial request. Since the original callback was already used in the initial exchange, and the callback is intended for one-time-use, a new one needs to be included in order to use the callback again.

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/654321",
            "nonce": "K82FX4T4LKLTI25DQFZC"
        }
    }
}
```

From here, the AS can determine that the client instance is asking for more than it was previously granted, but since the client instance has also provided a mechanism to interact with the RO, the AS can use that to gather the additional consent. The protocol continues as it would with a new request. Since the old access tokens are good for a subset of the rights requested here, the AS might decide to not revoke them. However, any access tokens granted after this update process are new access tokens and do not modify the rights of existing access tokens.

5.4. Canceling a Grant Request

If the client instance wishes to cancel an ongoing grant request, it makes an HTTP DELETE request to the continuation URI.

```
DELETE /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

If the request is successfully cancelled, the AS responds with an HTTP 202. The AS SHOULD revoke all associated access tokens.

6. Token Management

If an access token response includes the manage parameter as described in Section 3.2.1, the client instance MAY call this URI to manage the access token with any of the actions defined in the following sections: rotate and revoke. Other actions are undefined by this specification.

The access token being managed acts as the access element for its own management API. The client instance MUST present proof of an appropriate key along with the access token.

If the token is sender-constrained (i.e., not a bearer token), it MUST be sent with the appropriate binding for the access token (Section 7.2).

If the token is a bearer token, the client instance MUST present proof of the same key identified in the initial request (Section 2.3) as described in Section 7.3.

The AS MUST validate the proof and assure that it is associated with either the token itself or the client instance the token was issued to, as appropriate for the token's presentation type.

6.1. Rotating the Access Token

If the client instance has an access token and that access token expires, the client instance might want to rotate the access token. Rotating an access token consists of issuing a new access token in place of an existing access token, with the same rights and properties as the original token, apart from an updated expiration time.

To rotate an access token, the client instance makes an HTTP POST to the token management URI, sending the access token in the appropriate header and signing the request with the appropriate key.

```
POST /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The AS validates that the token presented is associated with the management URI, that the AS issued the token to the given client instance, and that the presented key is appropriate to the token.

Note that in many cases, the access token will have expired for regular use. To facilitate token rotation, the AS SHOULD honor the rotation request of the expired access token since it is likely that the client instance is attempting to refresh the expired token. To support this, the AS MAY allow a longer lifetime for token management compared to its use at an RS. An AS MUST NOT honor a rotation request for an access token that has been revoked or otherwise disabled.

If the token is validated and the key is appropriate for the request, the AS MUST invalidate the current access token associated with this URI, if possible. Note that stateless access tokens can make proactive revocation difficult within a system, see Section 12.29.

The AS responds with an HTTP 200 with a JSON body consisting of the rotated access token in the access_token field described in Section 3.2.1. The value of the access token MUST NOT be the same as the current value of the access token used to access the management API. The response MUST include an access token management URI, and the value of this URI MAY be different from the URI used by the client instance to make the rotation call. The client instance MUST use this new URI to manage the rotated access token.

The access rights in the access array for the rotated access token MUST be included in the response and MUST be the same as the token before rotation. If the client instance requires different access rights, the client instance can request a new access token by creating a new request (Section 2) or by updating an existing grant request (Section 5.3).

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "value": "FP6A8H6HY37MH13CK76LBZ6Y1UADG6VEUPEER5H2",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "expires_in": 3600,
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata"
        ]
    }
}
```

[[See issue #103 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/103)]]

6.2. Revoking the Access Token

If the client instance wishes to revoke the access token proactively, such as when a user indicates to the client instance that they no longer wish for it to have access or the client instance application detects that it is being uninstalled, the client instance can use the token management URI to indicate to the AS that the AS should invalidate the access token for all purposes.

The client instance makes an HTTP DELETE request to the token management URI, presenting the access token and signing the request with the appropriate key.

```
DELETE /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Signature-Input: sig1=...
Signature: sig1=...
```

If the key presented is associated with the token (or the client instance, in the case of a bearer token), the AS MUST invalidate the access token, if possible, and return an HTTP 204 response code.

```
204 No Content
```

Though the AS MAY revoke an access token at any time for any reason, the token management function is specifically for the client instance's use. If the access token has already expired or has been revoked through other means, the AS SHOULD honor the revocation request to the token management URI as valid, since the end result is still the token not being usable.

7. Securing Requests from the Client Instance

In GNAP, the client instance secures its requests to the AS and RS by presenting an access token, presenting proof of a key that it possesses (aka, a "key proof"), or both an access token and key proof together.

* When an access token is used with a key proof, this is a bound token request. This type of request is used for calls to the RS as well as the AS during negotiation.

* When a key proof is used with no access token, this is a non-authorized signed request. This type of request is used for calls to the AS to initiate a negotiation.

* When an access token is used with no key proof, this is a bearer token request. This type of request is used only for calls to the RS, and only with access tokens that are not bound to any key as described in Section 3.2.1.

* When neither an access token nor key proof are used, this is an unsecured request. This type of request is used optionally for calls to the RS as part of an RS-first discovery process as described in Section 9.1.

7.1. Key Formats

Several different places in GNAP require the presentation of key material by value. Proof of this key material MUST be bound to a request, the nature of which varies with the location in the protocol the key is used. For a key used as part of a client instance's initial request in Section 2.3, the key value is the client instance's public key, and proof of that key MUST be presented in that request. For a key used as part of an access token response in Section 3.2.1, the proof of that key MUST be used when presenting the access token.

A key presented by value MUST be a public key in at least one supported format. If a key is sent in multiple formats, all the key format values MUST be equivalent. Note that while most formats present the full value of the public key, some formats present a value cryptographically derived from the public key.

proof (string): The form of proof that the client instance will use when presenting the key. The valid values of this field and the processing requirements for each are detailed in Section 7.3. REQUIRED.

jwk (object): The public key and its properties represented as a JSON Web Key [RFC7517]. A JWK MUST contain the alg (Algorithm) and kid (Key ID) parameters. The alg parameter MUST NOT be "none". The x5c (X.509 Certificate Chain) parameter MAY be used to provide the X.509 representation of the provided public key. OPTIONAL.

cert (string): PEM serialized value of the certificate used to sign the request, with optional internal whitespace per [RFC7468]. The PEM header and footer are optionally removed. OPTIONAL.

cert#S256 (string): The certificate thumbprint calculated as per OAuth-MTLS [RFC8705] in base64 URL encoding. Note that this format does not include the full public key. OPTIONAL.

Additional key formats are defined in a registry TBD (Section 11).

This non-normative example shows a single key presented in multiple formats. This example key is intended to be used with the HTTP Message Signatures ({{httpsig-binding}}) proofing mechanism, as indicated by the httpsig value of the proof field.

```
   "key": {
       "proof": "httpsig",
       "jwk": {
           "kty": "RSA",
           "e": "AQAB",
           "kid": "xyz-1",
           "alg": "RS256",
           "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
       },
       "cert": "MIIEHDCCAwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
   }
```

7.1.1. Key References

Keys in GNAP can also be passed by reference such that the party receiving the reference will be able to determine the appropriate keying material for use in that part of the protocol.

```
       "key": "S-P4XJQ_RYJCRTSU1.63N3E"
```

Keys referenced in this manner MAY be shared symmetric keys. The key reference MUST NOT contain any unencrypted private or shared symmetric key information.

Keys referenced in this manner MUST be bound to a single proofing mechanism.

The means of dereferencing this value are out of scope for this specification. Commonly, key references are created by the AS and are not necessarily needed to be dereferencable by the client. These types of key references are an internal reference to the AS, such as an identifier of a record in a database. In other applications, it can be useful to use key references that are resolvable by both clients and ASs, which could be accomplished by e.g. a client publishing a public key at a URI. For interoperability, this method could later be described as an extension.

7.1.2. Key Protection

The security of GNAP relies on the cryptographic security of the keys themselves. When symmetric keys are used in GNAP, a key management system or secure key derivation mechanism MUST be used to supply the keys. Symmetric keys MUST NOT be a human memorable password or a value derived from one. Symmetric keys MUST NOT be passed by value from the client instance to the AS.

7.2. Presenting Access Tokens

The method the client instance uses to send an access token depends on whether the token is bound to a key, and if so which proofing method is associated with the key. This information is conveyed by the key parameter and the bearer flag in the single (Section 3.2.1) and multiple access tokens (Section 3.2.2) responses.

If the flags field does not contain the bearer flag and the key is absent, the access token MUST be sent using the same key and proofing mechanism that the client instance used in its initial request (or its most recent rotation).

If the flags field does not contain the bearer flag and the key value is an object as described in Section 7.1, the access token MUST be sent using the key and proofing mechanism defined by the value of the proof field within the key object.

The access token MUST be sent using the HTTP "Authorization" request header field and the "GNAP" authorization scheme along with a key proof as described in Section 7.3 for the key bound to the access token. For example, an "httpsig"-bound access token is sent as follows:

NOTE: '\' line wrapping per RFC 8792

```
GET /stuff HTTP/1.1
Host: resource.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=("@method" "@target-uri" "authorization")\
  ;created=1618884473;keyid="gnap-rsa"
Signature: sig1=:ThgXGQjGiJYQW8JYxcNypXk7wQWG8KZ6AtyKOrqNOkgoa8iWgm\
  feHLkRmT6BUj83DkLX84TQehhK3D5Lcgllhghuu2Pr3JmYVY7FFYwYAcfoISzVPKp\
  YyDbh/g34qOpFvlCYDgG94ZX16LAKlqYXWn5vYgealgm54zzCCnvyaLKViGVWz6PM\
  7rOIZqMQPOu6JceqdsiVn8xj2qTS9CWEmuJABtTnRoXNGVg8tUEQp7qt3F7tCI/AM\
  vHW4FAYrQbE47qQsjh4zPiES1EM+lHdA9fCE0OEsfabxB7Gr9GvkMyiApWTf/Zs45\
  IoJhr1OVtOCGVhEmoiNFreBTm7cTyTgg==:
```

If the flags field contains the bearer flag, the access token is a bearer token that MUST be sent using the Authorization Request Header Field method defined in [RFC6750].

```
Authorization: Bearer OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
```

The Form-Encoded Body Parameter and URI Query Parameter methods of [RFC6750] MUST NOT be used.

[[See issue #104 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/104)]]

The client software MUST reject as an error a situation where the flags field contains the bearer flag and the key field is present with any value.

7.3. Proving Possession of a Key with a Request

Any keys presented by the client instance to the AS or RS MUST be validated as part of the request in which they are presented. The type of binding used is indicated by the proof parameter of the key object in Section 7.1. Values defined by this specification are as follows:

"httpsig": HTTP Signing signature headers. See Section 7.3.1.

"mtls": Mutual TLS certificate verification. See Section 7.3.2.

"jwsd": A detached JWS signature header. See Section 7.3.3.

"jws": Attached JWS payload. See Section 7.3.4.

Additional proofing methods are defined by a registry TBD (Section 11).

All key binding methods used by this specification MUST cover all relevant portions of the request, including anything that would change the nature of the request, to allow for secure validation of the request. Relevant aspects include the URI being called, the HTTP method being used, any relevant HTTP headers and values, and the HTTP message body itself. The verifier of the signed message MUST validate all components of the signed message to ensure that nothing has been tampered with or substituted in a way that would change the nature of the request. Key binding method definitions SHOULD enumerate how these requirements are fulfilled.

When a key proofing mechanism is bound to an access token, the key being presented MUST be the key associated with the access token and the access token MUST be covered by the signature method of the proofing mechanism.

The key binding methods in this section MAY be used by other components making calls as part of GNAP, such as the extensions allowing the RS to make calls to the AS defined in [I-D.ietf-gnap-resource-servers]. To facilitate this extended use, the sections below are defined in generic terms of the "signer" and "verifier" of the HTTP message. In the core functions of GNAP, the "signer" is the client instance and the "verifier" is the AS or RS, as appropriate.

When used for delegation in GNAP, these key binding mechanisms allow the AS to ensure that the keys presented by the client instance in the initial request are in control of the party calling any follow-up or continuation requests. To facilitate this requirement, the continuation response (Section 3.1) includes an access token bound to the client instance's key (Section 2.3), and that key (or its most recent rotation) MUST be proved in all continuation requests Section 5. Token management requests Section 6 are similarly bound to either the access token's own key or, in the case of bearer tokens, the client instance's key.

[[See issue #105 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/105)]]

In the following sections, unless otherwise noted, the RS256 JOSE Signature Algorithm is applied using the following RSA key (presented here in JWK format):

NOTE: '\' line wrapping per RFC 8792

```
{
    "kid": "gnap-rsa",
    "p": "xS4-YbQ0SgrsmcA7xDzZKuVNxJe3pCYwdAe6efSy4hdDgF9-vhC5gjaRk\
        i1wWuERSMW4Tv44l5HNrL-Bbj_nCJxr_HAOaesDiPn2PnywwEfg3Nv95Nn-\
        eilhqXRaW-tJKEMjDHu_fmJBeemHNZI412gBnXdGzDVo22dvYoxd6GM",
    "kty": "RSA",
    "q": "rVdcT_uy-CD0GKVLGpEGRR7k4JO6Tktc8MEHkC6NIFXihk_6vAIOCzCD6\
        LMovMinOYttpRndKoGTNdJfWlDFDScAs8C5n2y1STCQPRximBY-bw39-aZq\
        JXMxOLyPjzuVgiTOCBIvLD6-8-mvFjXZk_eefD0at6mQ5qV3U1jZt88",
    "d": "FHlhdTF0ozTliDxMBffT6aJVKZKmbbFJOVNten9c3lXKB3ux3NAb_D2dB\
        7inp9EV23oWrDspFtvCvD9dZrXgRKMHofkEpo_SSvBZfgtH-OTkbY_TqtPF\
        FLPKAw0JX5cFPnn4Q2xE4n-dQ7tpRCKl59vZLHBrHShr90zqzFp0AKXU5fj\
        b1gC9LPwsFA2Fd7KXmI1drQQEVq9R-o18Pnn4BGQNQNjO_VkcJTiBmEIVT_\
        KJRPdpVJAmbgnYWafL_hAfeb_dK8p85yurEVF8nCK5oO3EPrqB7IL4UqaEn\
        5Sl3u0j8x5or-xrrAoNz-gdOv7ONfZY6NFoa-3f8q9wBAHUuQ",
    "e": "AQAB",
    "qi": "ogpNEkDKg22Rj9cDV_-PJBZaXMk66Fp557RT1tafIuqJRHEufSOYnsto\
        bWPJ0gHxv1gVJw3gm-zYvV-wTMNgr2wVsBSezSJjPSjxWZtmT2z68W1DuvK\
        kZy15vz7Jd85hmDlriGcXNCoFEUsGLWkpHH9RwPIzguUHWmTt8y0oXyI",
    "dp": "dvCKGI2G7RLh3WyjoJ_Dr6hZ3LhXweB3YcY3qdD9BnxZ71mrLiMQg4c_\
        EBnwqCETN_5sStn2cRc2JXnvLP3G8t7IFKHTT_i_TSTacJ7uT04MSa053Y3\
        RfwbvLjRNPR0UKAE3ZxROUoIaVNuU_6-QMf8-2ilUv2GIOrCN87gP_Vk",
    "alg": "RS256",
    "dq": "iMZmELaKgT9_W_MRT-UfDWtTLeFjIGRW8aFeVmZk9R7Pnyt8rNzyN-IQ\
        M40ql8u8J6vc2GmQGfokLlPQ6XLSCY68_xkTXrhoU1f-eDntkhP7L6XawSK\
        Onv5F2H7wyBQ75HUmHTg8AK2B_vRlMyFKjXbVlzKf4kvqChSGEz4IjQ",
    "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAt\
        YKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZGYX\
        jHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZx\
        e0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0\
        bunS0K3bA_3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kO\
        zywzwPTuq-cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
}
```

7.3.1. HTTP Message Signing

This method is indicated by httpsig in the proof field. The signer creates an HTTP Message Signature as described in [I-D.ietf-httpbis-message-signatures].

The covered components of the signature MUST include the following:

"@method": The method used in the HTTP request.

"@target-uri": The full request URI of the HTTP request.

When the message contains a request body, the covered components MUST also include the following:

"content-digest": The Content-Digest header as defined in [I-D.ietf-httpbis-digest-headers]. When the request message has a body, the signer MUST calculate this header value and the verifier MUST validate this field value. Use of content-encoding agnostic digest methods (such as sha-256) is RECOMMENDED.

When the request is bound to an access token, the covered components MUST also include the following:

"authorization": The Authorization header used to present the access token as discussed in Section 7.2.

Other message components MAY also be included.

If the signer's key presented is a JWK, the keyid parameter of the signature MUST be set to the kid value of the JWK, the signing algorithm used MUST be the JWS algorithm denoted by the key's alg field, and the explicit alg signature parameter MUST NOT be included.

In this example, the message body is the following JSON object:

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "PS512",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This body is hashed for the Content-Digest header using sha-256 into the following encoded value:

```
sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAGg=:
```

The HTTP message signature input string is calculated to be the following:

```
NOTE: '\' line wrapping per RFC 8792

"@method": POST
"@target-uri": https://server.example.com/gnap
"content-digest": \
  sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAGg=:
"content-length": 988
"content-type": application/json
"@signature-params": ("@method" "@target-uri" "content-digest" \
  "content-length" "content-type");created=1618884473;keyid="gnap-rsa"
```

This leads to the following full HTTP message request:

```
NOTE: '\' line wrapping per RFC 8792

POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 988
Content-Digest: sha-256=:q2XBmzRDCREcS2nWo/6LYwYyjrlN1bRfv+HKLbeGAG\
  g=:
Signature-Input: sig1=("@method" "@target-uri" "content-digest" \
  "content-length" "content-type");created=1618884473;keyid="gnap-rsa"
Signature: sig1=:EWJgAONk3D6542Scj8g51rYeMHw96cH2XiCMxcyL511wyemGcw\
  5PosYVO3eK+v+h1H+LiO4BjapL5ffZV+SgU8Q2v+qEDA4FrP0+/ni9W+lazjIrzNs\
  FAojwTlngMkAjZyDC/5+qUYB0KeEb4gnAhmuikv28DF30MT28yxCjeui2NGyzpPxB\
  cWk1K2Cxb6hS1WXUSZufFN9jOzrTg2c8/jcKkROKbLZLshF/oCuxAAgDabTqJy+qk\
  kz/Z/U5hI181qlTzNIYijnAvXzezlsLPZcMpJ1Au9APyBYAtDipAzyD6+IZl3rhzP\
  2leuCMCOvDxg9qA83LVtsqfjNJO+dEHA==:

{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
                "kid": "gnap-rsa",
                "kty": "RSA",
                "e": "AQAB",
                "alg": "PS512",
                "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
     YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
     YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
     ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
     3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
     N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
          }
        }
        "display": {
          "name": "My Client Display Name",
          "uri": "https://client.foo/"
        },
      }
   }
```

If the HTTP Message includes a message body, the verifier MUST calculate and verify the value of the Digest or Content-Digest header. The verifier MUST ensure that the signature covers all required message components. The verifier MUST validate the signature against the expected key of the signer.

7.3.2. Mutual TLS

This method is indicated by mtls in the proof field. The signer presents its TLS client certificate during TLS negotiation with the verifier.

In this example, the certificate is communicated to the application through the Client-Cert header from a TLS reverse proxy, leading to the following full HTTP request message:

```
   POST /gnap HTTP/1.1
   Host: server.example.com
   Content-Type: application/jose
   Content-Length: 1567
   Client-Cert: \
     :MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMDYxNDAyBgNVBAMM\
     K05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV6QzY2bVEwHhcN\
     MjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQDDCtOSVlNeUJq\
     c0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBIjANBgkqhkiG\
     9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT0VWtQBsmBB\
     kI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8I\
     kZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn11V2vxE4\
     1hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo+\
     uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKXfGhi3k\
     OzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0GCSqG\
     SIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/XsWfCE\
     wHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5NH9\
     W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeCgu\
     NMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHlU\
     fn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv\
     jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx:


   {
       "access_token": {
           "access": [
               "dolphin-metadata"
           ]
       },
       "interact": {
           "start": ["redirect"],
           "finish": {
               "method": "redirect",
               "uri": "https://client.foo/callback",
               "nonce": "VJLO6A4CAYLBXHTR0KRO"
           }
       },
       "client": {
         "key": {
           "proof": "mtls",
           "cert": "MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMD\
     YxNDAyBgNVBAMMK05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV\
     6QzY2bVEwHhcNMjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQD\
     DCtOSVlNeUJqc0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBI\
     jANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT\
     0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8\
     KowlyVy8IkZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn\
     11V2vxE41hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDad\
     z8BkPo+uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKX\
     fGhi3kOzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0\
     GCSqGSIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/Xs\
     WfCEwHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5\
     NH9W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeC\
     guNMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHl\
     Ufn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv\
     jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx"
         }
         "display": {
           "name": "My Client Display Name",
           "uri": "https://client.foo/"
         },
       },
       "subject": {
           "formats": ["iss_sub", "opaque"]
       }
   }
```

The verifier compares the TLS client certificate presented during mutual TLS negotiation to the expected key of the signer. Since the TLS connection covers the entire message, there are no additional requirements to check.

Note that in many instances, the verifier will not do a full certificate chain validation of the presented TLS client certificate, as the means of trust for this certificate could be in something other than a PKI system, such as a static registration or trust-on-first-use. See Section 12.17 and Section 12.18 for some additional considerations for this key proofing method.

7.3.3. Detached JWS

This method is indicated by jwsd in the proof field. A JWS [RFC7515] object is created as follows:

To protect the request, the JOSE header of the signature contains the following claims:

kid (string): The key identifier. REQUIRED if the key is presented in JWK format, this MUST be the value of the kid field of the key.

alg (string): The algorithm used to sign the request. MUST be appropriate to the key presented. If the key is presented as a JWK, this MUST be equal to the alg parameter of the key. MUST NOT be none. REQUIRED.

typ (string): The type header, value "gnap-binding+jwsd". REQUIRED.

htm (string): The HTTP Method used to make this request, as a case-sensitive ASCII string. Note that most public HTTP methods are in uppercase ASCII by convention. REQUIRED.

uri (string): The HTTP URI used for this request, including all path and query components and no fragment component. REQUIRED.

created (integer): A timestamp of when the signature was created, in integer seconds since UNIX Epoch. REQUIRED.

When the request is bound to an access token, the JOSE header MUST also include the following:

ath (string): The hash of the access token. The value MUST be the result of Base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token's value. REQUIRED.

If the HTTP request has a message body, such as an HTTP POST or PUT method, the payload of the JWS object is the Base64url encoding (without padding) of the SHA256 digest of the bytes of the body. If the request being made does not have a message body, such as an HTTP GET, OPTIONS, or DELETE method, the JWS signature is calculated over an empty payload.

The signer presents the signed object in compact form [RFC7515] in the Detached-JWS HTTP Header field.

In this example, the JOSE Header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding+jwsd",
    "created": 1618884475
}
```

The request body is the following JSON object:

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jwsd",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This is hashed to the following Base64 encoded value:

```
PGiVuOZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc
```

This leads to the following full HTTP request message:

```
NOTE: '\' line wrapping per RFC 8792

POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 983
Detached-JWS: eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0b\
  SI6IlBPU1QiLCJraWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3\
  NkIiwidXJpIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.PGiVuO\
  ZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc.fUq-SV-A1iFN2MwCRW_yolVtT2_\
  TZA2h5YeXUoi5F2Q2iToC0Tc4drYFOSHIX68knd68RUA7yHqCVP-ZQEd6aL32H69e\
  9zuMiw6O_s4TBKB3vDOvwrhYtDH6fX2hP70cQoO-47OwbqP-ifkrvI3hVgMX9TfjV\
  eKNwnhoNnw3vbu7SNKeqJEbbwZfpESaGepS52xNBlDNMYBQQXxM9OqKJaXffzLFEl\
  -Xe0UnfolVtBraz3aPrPy1C6a4uT7wLda3PaTOVtgysxzii3oJWpuz0WP5kRujzDF\
  wX_EOzW0jsjCSkL-PXaKSpZgEjNjKDMg9irSxUISt1C1T6q3SzRgfuQ


{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jwsd",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

When the verifier receives the Detached-JWS header, it MUST parse and validate the JWS object. The signature MUST be validated against the expected key of the signer. All required fields MUST be present and their values MUST be valid. If the HTTP message request contains a body, the verifier MUST calculate the hash of body just as the signer does, with no normalization or transformation of the request.

7.3.4. Attached JWS

This method is indicated by jws in the proof field. A JWS [RFC7515] object is created as follows:

To protect the request, the JWS header contains the following claims.

kid (string): The key identifier. REQUIRED if the key is presented in JWK format, this MUST be the value of the kid field of the key.

alg (string): The algorithm used to sign the request. MUST be appropriate to the key presented. If the key is presented as a JWK, this MUST be equal to the alg parameter of the key. MUST NOT be none. REQUIRED.

typ (string): The type header, value "gnap-binding+jwsd". REQUIRED.

htm (string): The HTTP Method used to make this request, as a case-sensitive ASCII string. (Note that most public HTTP methods are in uppercase.) REQUIRED.

uri (string): The HTTP URI used for this request, including all path and query components and no fragment component. REQUIRED.

created (integer): A timestamp of when the signature was created, in integer seconds since UNIX Epoch. REQUIRED.

When the request is bound to an access token, the JOSE header MUST also include the following:

ath (string): The hash of the access token. The value MUST be the result of Base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token's value. REQUIRED.

If the HTTP request has a message body, such as an HTTP POST or PUT method, the payload of the JWS object is the JSON serialized body of the request, and the object is signed according to JWS and serialized into compact form [RFC7515]. The signer presents the JWS as the body of the request along with a content type of application/jose. The verifier MUST extract the payload of the JWS and treat it as the request body for further processing.

If the request being made does not have a message body, such as an HTTP GET, OPTIONS, or DELETE method, the JWS signature is calculated over an empty payload and passed in the Detached-JWS header as described in Section 7.3.3.

In this example, the JOSE header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding+jwsd",
    "created": 1618884475
}
```

The request body, used as the JWS Payload, is the following JSON object:

NOTE: '\' line wrapping per RFC 8792

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "key": {
        "proof": "jws",
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    },
    "subject": {
        "formats": ["iss_sub", "opaque"]
    }
}
```

This leads to the following full HTTP request message:

NOTE: '\' line wrapping per RFC 8792

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/jose
Content-Length: 1047
```

```
eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0bSI6IlBPU1QiLCJ\
raWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3NkIiwidXJpIjoiaH\
R0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.CnsKICAgICJhY2Nlc3NfdG9r\
ZW4iOiB7CiAgICAgICAgImFjY2VzcyI6IFsKICAgICAgICAgICAgImRvbHBoaW4tbWV\
0YWRhdGEiCiAgICAgICAgXQogICAgfSwKICAgICJpbnRlcmFjdCI6IHsKICAgICAgIC\
Aic3RhcnQiOiBbInJlZGlyZWN0Il0sCiAgICAgICAgImZpbmlzaCI6IHsKICAgICAgI\
CAgICAgIm1ldGhvZCI6ICJyZWRpcmVjdCIsCiAgICAgICAgICAgICJ1cmkiOiAiaHR0\
cHM6Ly9jbGllbnQuZm9vL2NhbGxiYWNrIiwKICAgICAgICAgICAgIm5vbmNlIjogIlZ\
KTE82QTRDQVlMQlhIVFIwS1JPIgogICAgICAgIH0KICAgIH0sCiAgICAiY2xpZW50Ij\
ogewogICAgICAicHJvb2YiOiAiandzIiwKICAgICAgImtleSI6IHsKICAgICAgICAia\
ndrIjogewogICAgICAgICAgICAia2lkIjogImduYXAtcnNhIiwKICAgICAgICAgICAg\
Imt0eSI6ICJSU0EiLAogICAgICAgICAgICAiZSI6ICJBUUFCIiwKICAgICAgICAgICA\
gImFsZyI6ICJSUzI1NiIsCiAgICAgICAgICAgICJuIjogImhZT0otWE9LSVNkTU1TaG\
5fRzRXOW0yMG1UMFZXdFFCc21CQmtJMmNtUnQ0QWk4QmZZZEhzRnpBdFlLT2pwQlIxU\
nBLcEptVkt4SUdOeTBnNlozYWQyWFlzaDhLb3dseVZ5OElrWjhOTXdTcmNVSUJaR1lY\
akhwd2p6dmZHdlhIXzVLSmxuUjNfdVJVcDRaNFVqazJiQ2FLZWdEbjExVjJ2eEU0MWh\
xYVBVbmhSWnhlMGpSRVRkZHpzRTNtdTFTSzhkVENST2p3VWwxNG1VTm84aVRyVG00bj\
BxRGFkejhCa1BvLXV2NEJDMGJ1blMwSzNiQV8zVWdWcDd6QmxRRm9GbkxUTzJ1V3Bfb\
XVMRVdHbDY3Z0JxOU1PM2JyS1hmR2hpM2tPenl3endQVHVxLWNWUUR5RU43YUwwU3hD\
YjNIYzRJZHFEYU1nOHFIVXlPYnBQaXREUSIKICAgICAgICB9CiAgICAgIH0KICAgICA\
gImRpc3BsYXkiOiB7CiAgICAgICAgIm5hbWUiOiAiTXkgQ2xpZW50IERpc3BsYXkgTm\
FtZSIsCiAgICAgICAgInVyaSI6ICJodHRwczovL2NsaWVudC5mb28vIgogICAgICB9L\
AogICAgfSwKICAgICJzdWJqZWN0IjogewogICAgICAgICJmb3JtYXRzIjogWyJpc3Nf\
c3ViIiwgIm9wYXF1ZSJdCiAgICB9Cn0K.MwNoVMQp5hVxI0mCs9LlOUdFtkDXaA1_eT\
vOXq7DOGrtDKH7q4vP2xUq3fH2jRAZqnobo0WdPP3eM3NH5QUjW8pa6_QpwdIWkK7r-\
u_52puE0lPBp7J4U2w4l9gIbg8iknsmWmXeY5F6wiGT8ptfuEYGgmloAJd9LIeNvD3U\
LW2h2dz1Pn2eDnbyvgB0Ugae0BoZB4f69fKWj8Z9wvTIjk1LZJN1PcL7_zT8Lrlic9a\
PyzT7Q9ovkd1s-4whE7TrnGUzFc5mgWUn_gsOpsP5mIIljoEEv-FqOW2RyNYulOZl0Q\
8EnnDHV_vPzrHlUarbGg4YffgtwkQhdK72-JOxYQ
```

When the verifier receives an attached JWS request, it MUST parse and validate the JWS object. The signature MUST be validated against the expected key of the signer. All required fields MUST be present and their values MUST be valid. If the HTTP message request contains a body, the verifier MUST decode the payload of the JWS object and treat this as the HTTP message body.

8. Resource Access Rights

GNAP provides a rich structure for describing the protected resources hosted by RSs and accessed by client software. This structure is used when the client instance requests an access token (Section 2.1) and when an access token is returned (Section 3.2).

The root of this structure is a JSON array. The elements of the JSON array represent rights of access that are associated with the the access token. The resulting access is the union of all elements within the array.

The access associated with the access token is described using objects that each contain multiple dimensions of access. Each object contains a REQUIRED type property that determines the type of API that the token is used for.

type (string): The type of resource request as a string. This field MAY define which other fields are allowed in the request object. REQUIRED.

The value of the type field is under the control of the AS. This field MUST be compared using an exact byte match of the string value against known types by the AS. The AS MUST ensure that there is no collision between different authorization data types that it supports. The AS MUST NOT do any collation or normalization of data types during comparison. It is RECOMMENDED that designers of general-purpose APIs use a URI for this field to avoid collisions between multiple API types protected by a single AS.

While it is expected that many APIs will have their own properties, a set of common properties are defined here. Specific API implementations SHOULD NOT re-use these fields with different semantics or syntax. The available values for these properties are determined by the API being protected at the RS. All values are OPTIONAL at the discretion of the API definition.

actions (array of strings): The types of actions the client instance will take at the RS as an array of strings. For example, a client instance asking for a combination of "read" and "write" access.

locations (array of strings): The location of the RS as an array of strings. These strings are typically URIs identifying the location of the RS.

datatypes (array of strings): The kinds of data available to the client instance at the RS's API as an array of strings. For example, a client instance asking for access to raw "image" data and "metadata" at a photograph API.

identifier (string): A string identifier indicating a specific resource at the RS. For example, a patient identifier for a medical API or a bank account number for a financial API.

privileges (array of strings): The types or levels of privilege being requested at the resource. For example, a client instance asking for administrative level access, or access when the resource owner is no longer online.

The following non-normative example is describing three kinds of access (read, write, delete) to each of two different locations and two different data types (metadata, images) for a single access token using the fictitious photo-api type definition.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    }
]
```

The access requested for a given object when using these fields is the cross-product of all fields of the object. That is to say, the object represents a request for all actions listed to be used at all locations listed for all possible datatypes listed within the object. Assuming the request above was granted, the client instance could assume that it would be able to do a read action against the images on the first server as well as a delete action on the metadata of the second server, or any other combination of these fields, using the same access token.

To request a different combination of access, such as requesting one of the possible actions against one of the possible locations and a different choice of possible actions against a different one of the possible locations, the client instance can include multiple separate objects in the resources array. The following non-normative example uses the same fictitious photo-api type definition to request a single access token with more specifically targeted access rights by using two discrete objects within the request.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read"
        ],
        "locations": [
            "https://server.example.net/"
        ],
        "datatypes": [
            "images"
        ]
    },
    {
        "type": "photo-api",
        "actions": [
            "write",
            "delete"
        ],
        "locations": [
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata"
        ]
    }
]
```

The access requested here is for read access to images on one server while simultaneously requesting write and delete access for metadata on a different server, but importantly without requesting write or delete access to images on the first server.

It is anticipated that API designers will use a combination of common fields defined in this specification as well as fields specific to the API itself. The following non-normative example shows the use of both common and API-specific fields as part of two different fictitious API type values. The first access request includes the actions, locations, and datatypes fields specified here as well as the API-specific geolocation field. The second access request includes the actions and identifier fields specified here as well as the API-specific currency field.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ],
        "geolocation": [
            { lat: -32.364, lng: 153.207 },
            { lat: -35.364, lng: 158.207 }
        ]
    },
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    }
]
```

If this request is approved, the resulting access token (Section 3.2.1)'s access rights will be the union of the requested types of access for each of the two APIs, just as above.

8.1. Requesting Resources By Reference

Instead of sending an object describing the requested resource (Section 8), access rights MAY be communicated as a string known to the AS or RS representing the access being requested. Each string SHOULD correspond to a specific expanded object representation at the AS.

```
"access": [
    "read", "dolphin-metadata", "some other thing"
]
```

This value is opaque to the client instance and MAY be any valid JSON string, and therefore could include spaces, unicode characters, and properly escaped string sequences. However, in some situations the value is intended to be seen and understood by the client software's developer. In such cases, the API designer choosing any such human-readable strings SHOULD take steps to ensure the string values are not easily confused by a developer, such as by limiting the strings to easily disambiguated characters.

This functionality is similar in practice to OAuth 2.0's scope parameter [RFC6749], where a single string represents the set of access rights requested by the client instance. As such, the reference string could contain any valid OAuth 2.0 scope value as in Appendix D.5. Note that the reference string here is not bound to the same character restrictions as in OAuth 2.0's scope definition.

A single access array MAY include both object-type and string-type resource items. In this non-normative example, the client instance is requesting access to a photo-api and financial-transaction API type as well as the reference values of read, dolphin-metadata, and some other thing.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    },
    "read",
    "dolphin-metadata",
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    },
    "some other thing"
]
```

The requested access is the union of all elements of the array, including both objects and reference strings.

9. Discovery

By design, the protocol minimizes the need for any pre-flight discovery. To begin a request, the client instance only needs to know the endpoint of the AS and which keys it will use to sign the request. Everything else can be negotiated dynamically in the course of the protocol.

However, the AS can have limits on its allowed functionality. If the client instance wants to optimize its calls to the AS before making a request, it MAY send an HTTP OPTIONS request to the grant request endpoint to retrieve the server's discovery information. The AS MUST respond with a JSON document with Content-Type application/json containing a single object with the following information:

grant_request_endpoint (string): The location of the AS's grant request endpoint. The location MUST be a URL [RFC3986] with a scheme component that MUST be https, a host component, and optionally, port, path and query components and no fragment components. This URL MUST match the URL the client instance used to make the discovery request. REQUIRED.

interaction_start_modes_supported (array of strings): A list of the AS's interaction start methods. The values of this list correspond to the possible values for the interaction start section (Section 2.5.1) of the request. OPTIONAL.

interaction_finish_methods_supported (array of strings): A list of the AS's interaction finish methods. The values of this list correspond to the possible values for the method element of the interaction finish section (Section 2.5.2) of the request. OPTIONAL.

key_proofs_supported (array of strings): A list of the AS's supported key proofing mechanisms. The values of this list correspond to possible values of the proof field of the key section (Section 7.1) of the request. OPTIONAL.

sub_id_formats_supported (array of strings): A list of the AS's supported subject identifier formats. The values of this list correspond to possible values of the subject identifier section (Section 2.2) of the request. OPTIONAL.

assertion_formats_supported (array of strings): A list of the AS's supported assertion formats. The values of this list correspond to possible values of the subject assertion section (Section 2.2) of the request. OPTIONAL.

The information returned from this method is for optimization purposes only. The AS MAY deny any request, or any portion of a request, even if it lists a capability as supported. For example, a given client instance can be registered with the mtls key proofing mechanism, but the AS also returns other proofing methods from the discovery document, then the AS will still deny a request from that client instance using a different proofing mechanism.

9.1. RS-first Method of AS Discovery

If the client instance calls an RS without an access token, or with an invalid access token, the RS MAY respond to the client instance with an authentication header indicating that GNAP needs to be used to access the resource. The address of the GNAP endpoint MUST be sent in the "as_uri" parameter. The RS MAY additionally return a resource reference that the client instance MAY use in its access token request. This resource reference MUST be sufficient for at least the action the client instance was attempting to take at the RS and MAY be more powerful. The means for the RS to determine the resource reference are out of scope of this specification, but some dynamic methods are discussed in [I-D.ietf-gnap-resource-servers]. The content of the resource reference is opaque to the client instance.

```
NOTE: '\' line wrapping per RFC 8792

WWW-Authenticate: \
  GNAP as_uri=https://server.example/tx,access=FWWIKYBQ6U56NL1
```

The client instance then makes a request to the "as_uri" as described in Section 2, with the value of "access" as one of the members of the access array in the access_token portion of the request. The client instance MAY request additional resources and other information. The client instance MAY request multiple access tokens.

In this non-normative example, the client instance is requesting a single access token using the resource reference FWWIKYBQ6U56NL1 received from the RS in addition to the dolphin-metadata resource reference that the client instance has been configured with out of band.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "FWWIKYBQ6U56NL1",
            "dolphin-metadata"
        ]
    },
    "client": "KHRS6X63AJ7C7C4AZ9AO"
}
```

If issued, the resulting access token would contain sufficient access to be used at both referenced resources.

10. Acknowledgements

The editors would like to thank the feedback of the following individuals for their reviews, implementations, and contributions: Åke Axeland, Aaron Parecki, Adam Omar Oueidat, Andrii Deinega, Annabelle Backman, Dick Hardt, Dmitri Zagidulin, Dmitry Barinov, Fabien Imbault, Florian Helmschmidt, Francis Pouatcha, George Fletcher, Haardik Haardik, Hamid Massaoud, Jacky Yuan, Joseph Heenan, Justin Richer, Kathleen Moriarty, Mike Jones, Mike Varley, Nat Sakimura, Takahiko Kawasaki, Takahiro Tsuchiya.

The editors would also like to thank the GNAP working group design team of Kathleen Moriarty, Fabien Imbault, Dick Hardt, Mike Jones, and Justin Richer, who incorporated elements from the XAuth and XYZ proposals to create the first version of this document.

In addition, the editors would like to thank Aaron Parecki and Mike Jones for insights into how to integrate identity and authentication systems into the core protocol, and Justin Richer and Dick Hardt for the use cases, diagrams, and insights provided in the XYZ and XAuth proposals that have been incorporated here. The editors would like to especially thank Mike Varley and the team at SecureKey for feedback and development of early versions of the XYZ protocol that fed into this standards work.

11. IANA Considerations

[[TBD: There are a lot of items in the document that are expandable through the use of value registries.]]

12. Security Considerations

12.1. TLS Protection in Transit

All requests in GNAP have to be made over TLS or equivalent as outlined in [BCP195] to protect the contents of the request and response from manipulation and interception by an attacker. This includes all requests from a client instance to the AS, all requests from the client instance to an RS, any requests back to a client instance such as the push-based interaction finish method, and any back-end communications such as from an RS to an AS as described in [I-D.ietf-gnap-resource-servers]. Additionally, all requests between a browser and other components, such as during redirect-based interaction, need to be made over TLS or use equivalent protection.

Even though requests from the client instance to the AS are signed, the signature method alone does not protect the request from interception by an attacker. TLS protects the response as well as the request, preventing an attacker from intercepting requested information as it is returned. This is particularly important in the core protocol for security artifacts such as nonces and for personal information such as subject information.

The use of key-bound access tokens does not negate the requirement for protecting calls to the RS with TLS. While the keys and signatures associated a bound access token will prevent an attacker from using a stolen token, without TLS an attacker would be able to watch the data being sent to the RS and returned from the RS during legitimate use of the client instance under attack. Additionally, without TLS an attacker would be able to profile the calls made between the client instance and RS, possibly gaining information about the functioning of the API between the client software and RS software that would be otherwise unknown to the attacker.

TLS or equivalent protection also needs to be used between the browser and any other components. This applies during initial redirects to an AS's components during interaction, during any interaction with the resource owner, and during any redirect back to the client instance. Without TLS protection on these portions of the process, an attacker could wait for a valid request to start and then take over the resource owner's interaction session.

12.2. Signing Requests from the Client Software

Even though all requests in GNAP need to be transmitted over TLS or its equivalent, the use of TLS alone is not sufficient to protect all parts of a multi-party and multi-stage protocol like GNAP, and TLS is not targeted at tying multiple requests to each other over time. To account for this, GNAP makes use of message-level protection and key presentation mechanisms that strongly associate a request with a key held by the client instance (see Section 7).

During the initial request from a client instance to the AS, the client instance has to identify and prove possession of a cryptographic key. If the key is known to the AS, such as if it is previously registered or dereferenceable to a trusted source, the AS can associate a set of policies to the client instance identified by the key. Without the requirement that the client instance prove that it holds that key, the AS could not trust that the connection came from any particular client and could not apply any associated policies.

Even more importantly, the client instance proving possession of a key on the first request allows the AS to associate future requests with each other. The access token used for grant continuation is bound to the same key and proofing mechanism used by the client instance in its initial request, which means that the client instance needs to prove possession of that same key in future requests allowing the AS to be sure that the same client instance is executing the follow-ups for a given ongoing grant request. Therefore, the AS has to ensure that all subsequent requests for a grant are associated with the same key that started the grant, or the most recent rotation of that key. This need holds true even if the initial key is previously unknown to the AS, such as would be the case when a client instance creates an ephemeral key for its request. Without this ongoing association, an attacker would be able to impersonate a client instance in the midst of a grant request, potentially stealing access tokens and subject information with impunity.

Additionally, all access tokens in GNAP default to be associated with the key that was presented during the grant request that created the access token. This association allows an RS to know that the presenter of the access token is the same party that the token was issued to, as identified by their keys. While non-bound bearer tokens are an option in GNAP, these types of tokens have their own tradeoffs discussed elsewhere in this section.

TLS functions at the socket layer, ensuring that only the parties on either end of that socket connection can read the information passed along that connection. Each time a new socket connection is made, such as for a new HTTP request, a new trust is re-established that is unrelated to previous connections. As such, it is not possible with TLS alone to know that the same party is making a set of calls, and therefore TLS alone cannot provide the continuity of security needed for GNAP. However, mutual TLS (MTLS) does provide such security characteristics through the use of the TLS client certificate, and thus MTLS is acceptable as a key-presentation mechanism when applied as described in Section 7.3.2.

12.3. Protection of Client Instance Key Material

Client instances are identified by their unique keys, and anyone with access to a client instance's key material will be able to impersonate that client instance to all parties. This is true for both calls to the AS as well as calls to an RS using a key-bound access token.

Different types of client software have different methods available for creating, managing, and registering keys. GNAP explicitly allows for ephemeral clients, such as SPAs, and single-user clients, such as mobile applications, to create and present their own keys during the initial grant request. The client software can securely generate a keypair on-device and present the public key, along with proof of holding that public key, to the AS as part of the initial request. To facilitate trust in these ephemeral keys, GNAP further allows for an extensible set of client information to be passed with the request. This information can include device posture and third-party attestations of the client software's provenance and authenticity, depending on the needs and capabilities of the client software and its deployment.

From GNAP's perspective, each distinct key is a different client instance. However, multiple client instances can be grouped together by an AS policy and treated similarly to each other. For instance, if an AS knows of several different keys for different servers within a cluster, the AS can decide that authorization of one of these servers applies to all other servers within the cluster. An AS that chooses to do this needs to be careful with how it groups different client keys together in its policy, since the breach of one instance would have direct effects on the others in the cluster.

Additionally, if an end user controls multiple instances of a single type of client software, such as having an application installed on multiple devices, each of these instances is expected to have a separate key and be issued separate access tokens. However, if the AS is able to group these separate instances together as described above, it can streamline the authorization process for new instances of the same client software. For example, if two client instances can present proof of a valid installation of a piece of client software, the AS would be able to associate the approval of the first instance of this software to all related instances. The AS could then choose to bypass an explicit prompt of the resource owner for approval during authorization, since such approval has already been given. An AS doing such a process would need to take assurance measures that the different instances are in fact correlated and authentic, as well as ensuring the expected resource owner is in control of the client instance.

Finally, if multiple instances of client software each have the same key, then from GNAP's perspective, these are functionally the same client instance as GNAP has no reasonable way to differentiate between them. This situation could happen if multiple instances within a cluster can securely share secret information among themselves. Even though there are multiple copies of the software, the shared key makes these copies all present as a single instance. It is considered bad practice to share keys between copies of software unless they are very tightly integrated with each other and can be closely managed. It is particularly bad practice to allow an end user to copy keys between client instances and to willingly use the same key in multiple instances.

12.4. Protection of Authorization Server

The AS performs critical functions in GNAP, including authenticating client software, managing interactions with end users to gather consent and provide notice, and issuing access tokens for client instances to present to resource servers. As such, protecting the AS is central to any GNAP deployment.

If an attacker is able to gain control over an AS, they would be able to create fraudulent tokens and manipulate registration information to allow for malicious clients. These tokens and clients would be trusted by other components in the ecosystem under the protection of the AS.

If the AS is using signed access tokens, an attacker in control of the AS's signing keys would be able to manufacture fraudulent tokens for use at RS's under the protection of the AS.

If an attacker is able to impersonate an AS, they would be able to trick legitimate client instances into making signed requests for information which could potentially be proxied to a real AS. To combat this, all communications to the AS need to be made over TLS or its equivalent, and the software making the connection has to validate the certificate chain of the host it is connecting to.

Consequently, protecting, monitoring, and auditing the AS is paramount to preserving the security of a GNAP-protected ecosystem.

12.5. Symmetric and Asymmetric Client Instance Keys

The cryptographic methods used by GNAP for key-proofing can support both asymmetric and symmetric cryptography, and can be extended to use a wide variety of mechanisms. While symmetric cryptographic systems have some benefits in speed and simplicity, they have a distinct drawback that both parties need access to the same key in order to do both signing and verification of the message. This means that when the client instance calls the AS to request a token, the AS needs to know the exact value of the client instance's key (or be able to derive it) in order to validate the key proof signature. With asymmetric keys, the client needs only to send its public key to the AS to allow for verification that the client holds the associated private key, regardless of whether that key was pre-registered or not with the AS.

When used to bind to an access token, a key value must be known by the RS in order to validate the proof signature on the request. Common methods for communicating these proofing keys include putting information in a structured access token and allowing the RS to look up the associated key material against the value of the access token. With symmetric cryptography, both of these methods would expose the signing key to the RS, and in the case of an structured access token, potentially to any party that can see the access token itself unless the token's payload has been encrypted. Any of these parties would then be able to make calls using the access token by creating a valid signature. With asymmetric cryptography, the RS only needs to know the public key associated with the token in order to validate, and therefore cannot create any new calls.

Symmetric keys also have the expected advantage of providing better protection against quantum threats in the future. Also, these types of keys (and their secure derivations) are widely supported among many cloud-based key management systems.

While both signing approaches are allowed, GNAP treats these two classes of keys somewhat differently. Only the public portion of asymmetric keys are allowed to be sent by value in requests to the AS when establishing a connection. Since sending a symmetric key (or the private portion of an asymmetric key) would expose the signing material to any parties on the request path, including any attackers, sending these kinds of keys is prohibited. Symmetric keys can still be used by client instances, but only a reference to the key and not its value can be sent. This allows the AS to use pre-registered symmetric keys as well as key derivation schemes to take advantage of symmetric cryptography but without requiring key distribution at runtime, which would expose the keys in transit.

Both the AS and client software can use systems such as hardware security modules to strengthen their key security storage and generation for both asymmetric and symmetric keys (see also Section 7.1.2).

12.6. Generation of Access Tokens

The content of access tokens need to be such that only the generating AS would be able to create them, and the contents cannot be manipulated by an attacker to gain different or additional access rights.

One method for accomplishing this is to use a cryptographically random value for the access token, generated by the AS using a secure randomization function with sufficiently high entropy. The odds of an attacker guessing the output of the randomization function to collide with a valid access token are exceedingly small, and even then the attacker would not have any control over what the access token would represent since that information would be held close by the AS.

Another method for accomplishing this is to use a structured token that is cryptographically signed. In this case, the payload of the access token declares to the RS what the token is good for, but the signature applied by the AS during token generation covers this payload. Only the AS can create such a signature and therefore only the AS can create such a signed token. The odds of an attacker being able to guess a signature value with a useful payload are exceedingly small. This technique only works if all targeted RS's check the signature of the access token. Any RS that does not validate the signature of all presented tokens would be susceptible to injection of a modified or falsified token. Furthermore, an AS has to carefully protect the keys used to sign access tokens, since anyone with access to these signing keys would be able to create seemingly-valid access tokens using them.

12.7. Bearer Access Tokens

Bearer access tokens can be used by any party that has access to the token itself, without any additional information. As a natural consequence, any RS that a bearer token is presented to has the technical capability of presenting that bearer token to another RS, as long as the token is valid. It also means that any party that is able capture of the token value in storage or in transit is able to use the access token. While bearer tokens are inherently simpler, this simplicity has been misapplied and abused in making needlessly insecure systems.

In GNAP, key-bound access tokens are the default due to their higher security properties. While bearer tokens can be used in GNAP, their use should be limited to cases where the simplicity benefits outweigh the significant security downsides.

12.8. Key-Bound Access Tokens

Key-bound access tokens, as the name suggests, are bound to a specific key and must be presented along with proof of that key during use. The key itself is not presented at the same time as the token, so even if a token value is captured, it cannot be used to make a new request. This is particularly true for an RS, which will see the token value but will not see the keys used to make the request.

Key-bound access tokens provide this additional layer of protection only when the RS checks the signature of the message presented with the token. Acceptance of an invalid presentation signature, or failure to check the signature entirely, would allow an attacker to make calls with a captured access token without having access to the related signing key material.

In addition to validating the signature of the presentation message itself, the RS also needs to ensure that the signing key used is appropriate for the presented token. If an RS does not ensure that the right keys were used to sign a message with a specific token, an attacker would be able to capture an access token and sign the request with their own keys, thereby negating the benefits of using key-bound access tokens.

The RS also needs to ensure that sufficient portions of the message are covered by the signature. Any items outside the signature could still affect the API's processing decisions, but these items would not be strongly bound to the token presentation. As such, an attacker could capture a valid request, then manipulate portions of the request outside of the signature envelope in order to cause unwanted actions at the protected API.

Some key-bound tokens are susceptible to replay attacks, depending on the details of the signing method used. If a signature method covers only portions of a given request, that same signature proof can be used by an attacker to make a similar call, potentially even varying elements that are outside of the protection of the signature. Key proofing mechanisms used with access tokens therefore need to use replay protection mechanisms covered under the signature such as a per-message nonce, a reasonably short time validity window, or other uniqueness constraints. The details of using these will vary depending on the key proofing mechanism in use, but for example, HTTP Message Signatures has both a created and nonce signature parameter as well as the ability to cover significant portions of the HTTP message.

12.9. Exposure of End-user Credentials to Client Instance

As a delegation protocol, one of the main goals of GNAP is to prevent the client software from being exposed to any credentials or information about the end user or resource owner as a requirement of the delegation process. By using the variety of interaction mechanisms, the resource owner can interact with the AS without ever authenticating to the client software, and without the client software having to impersonate the resource owner through replay of their credentials.

Consequently, no interaction methods defined in the GNAP core require the end user to enter their credentials, but it is technologically possible for an extension to be defined to carry such values. Such an extension would be dangerous as it would allow rogue client software to directly collect, store, and replay the end user's credentials outside of any legitimate use within a GNAP request.

The concerns of such an extension could be mitigated through use of a challenge and response unlocked by the end user's credentials. For example, the AS presents a challenge as part of an interaction start method, and the client instance signs that challenge using a key derived from a password presented by the end user. It would be possible for the client software to collect this password in a secure software enclave without exposing the password to the rest of the client software or putting it across the wire to the AS. The AS can validate this challenge response against a known password for the identified end user. While an approach such as this does not remove all of the concerns surrounding such a password-based scheme, it is at least possible to implement in a more secure fashion than simply collecting and replaying the password. Even so, such schemes should only ever be used by trusted clients due to the ease of abusing them.

12.10. Mixing Up Authorization Servers

If a client instance is able to work with multiple AS's simultaneously, it is more possible for an attacker to add a compromised AS to the client instance's configuration and cause the client software to start a request at the compromised AS. This AS could then proxy the client's request to a valid AS in order to attempt to get the resource owner to approve access for the legitimate client instance.

A client instance needs to always be aware of which AS it is talking to throughout a grant process, and ensure that any callback for one AS does not get conflated with the callback to different AS. The interaction finish hash calculate allows a client instance to protect against this kind of substitution, but only if the client instance validates the hash. If the client instance does not use an interaction finish method or does not check the interaction finish hash value, the compromised AS can be granted a valid access token on behalf of the resource owner. See [AXELAND2021] for details of one such attack, which has been since addressed in this document by including the grant endpoint in the interaction hash calculation. The client instance still needs to validate the hash for the attack to be prevented.

12.11. Processing of Client-Presented User Information

GNAP allows the client instance to present assertions and identifiers of the current user to the AS as part of the initial request. This information should only ever be taken by the AS as a hint, since the AS has no way to tell if the represented person is present at the client software, without using an interaction mechanism. This information does not guarantee the given user is there, but it does constitute a statement by the client software that the AS can take into account.

For example, if a specific user is claimed to be present prior to interaction, but a different user is shown to be present during interaction, the AS can either determine this to be an error or signal to the client instance through returned subject information that the current user has changed from what the client instance thought. This user information can also be used by the AS to streamline the interaction process when the user is present. For example, instead of having the user type in their account identifier during interaction at a redirected URI, the AS can immediately challenge the user for their account credentials. Alternatively, if an existing session is detected, the AS can determine that it matches the identifier provided by the client and subsequently skip an explicit authentication event by the resource owner.

In cases where the AS trusts the client software more completely, due to policy or by previous approval of a given client instance, the AS can take this user information as a statement that the user is present and could issue access tokens and release subject information without interaction. The AS should only take such action in very limited circumstances, as a client instance could assert whatever it likes for the user's identifiers in its request.

When a client instance presents an assertion to the AS, the AS needs to evaluate that assertion. Since the AS is unlikely to be the intended audience of an assertion held by the client software, the AS will need to evaluate the assertion in a different context. Even in this case, the AS can still evaluate that the assertion was generated by a trusted party, was appropriately signed, and is within any time validity windows stated by the assertion. If the client instance's audience identifier is known to the AS and can be associated with the client instance's presented key, the AS can also evaluate that the appropriate client instance is presenting the claimed assertion. All of this will prevent an attacker from presenting a manufactured assertion, or one captured from an untrusted system. However, without validating the audience of the assertion, a captured assertion could be presented by the client instance to impersonate a given end user. In such cases, the assertion offers little more protection than a simple identifier would.

A special case exists where the AS is the generator of the assertion being presented by the client instance. In these cases, the AS can validate that it did issue the assertion and it is associated with the client instance presenting the assertion.

12.12. Client Instance Pre-registration

Each client instance is identified by its own unique key, and for some kinds of client software such as a web server or backend system, this identification can be facilitated by registering a single key for a piece of client software ahead of time. This registration can be associated with a set of display attributes to be used during the authorization process, identifying the client software to the user. In these cases, it can be assumed that only one instance of client software will exist, likely to serve many different users.

A client's registration record needs to include its identifying key. Furthermore, it is the case that any clients using symmetric cryptography for key proofing mechanisms need to have their keys pre-registered. The registration should also include any information that would aid in the authorization process, such as a display name and logo. The registration record can also limit a given client to ask for certain kinds of information and access, or be limited to specific interaction mechanisms at runtime.

It also is sensible to pre-register client instances when the software is acting autonomously, without the need for a runtime approval by a resource owner or any interaction with an end user. In these cases, an AS needs to rest on the trust decisions that have been determined prior to runtime in determining what rights and tokens to grant to a given client instance.

However, it does not make sense to pre-register many types of clients. Single-page applications (SPAs) and mobile/desktop applications in particular present problems with pre-registration. For SPAs, the instances are ephemeral in nature and long-term registration of a single instance leads to significant storage and management overhead at the AS. For mobile applications, each installation of the client software is a separate instance, and sharing a key among all instances would be detrimental to security as the compromise of any single installation would compromise all copies for all users.

An AS can treat these classes of client software differently from each other, perhaps by allowing access to certain high-value APIs only to pre-registered known clients, or by requiring an active end user delegation of authority to any client software not pre-registered.

An AS can also provide warnings and caveats to resource owners during the authorization process, allowing the user to make an informed decision regarding the software they are authorizing. For example, if the AS has done vetting of the client software and this specific instance, it can present a different authorization screen compared to a client instance that is presenting all of its information at runtime.

12.13. Client Instance Impersonation

If client instances are allowed to set their own user-facing display information, such as a display name and website URL, a malicious client instance could impersonate legitimate client software for the purposes of tricking users into authorizing the malicious client.

Requiring clients to pre-register does not fully mitigate this problem since many pre-registration systems have self-service portals for management of client registration, allowing authenticated developers to enter self-asserted information into the management portal.

An AS can mitigate this by actively filtering all self-asserted values presented by client software, both dynamically as part of GNAP and through a registration portal, to limit the kinds of impersonation that would be done.

An AS can also warn the resource owner about the provenance of the information it is displaying, allowing the resource owner to make a more informed delegation decision. For example, an AS can visually differentiate between a client instance that can be traced back to a specific developer's registration and an instance that has self-asserted its own key and display information.

12.14. Interception of Information in the Browser

Most information passed through the web-browser is susceptible to interception and possible manipulation by elements within the browser such as scripts loaded within pages. Information in the URI is exposed through browser and server logs, and can also leak to other parties through HTTP Referer headers.

GNAP's design limits the information passed directly through the browser, allowing for opaque URIs in most circumstances. For the redirect-based interaction finish mechanism, named query parameters are used to carry unguessable opaque values. For these, GNAP requires creation and validation of a cryptographic hash to protect the query parameters added to the URI and associate them with an ongoing grant process. The client instance has to properly validate this hash to prevent an attacker from injecting an interaction reference intended for a different AS or client instance.

Several interaction start mechanisms use URIs created by the AS and passed to the client instance. While these URIs are opaque to the client instance, it's possible for the AS to include parameters, paths, and other pieces of information that could leak security data or be manipulated by a party in the middle of the transaction.

12.15. Callback URI Manipulation

The callback URI used in interaction finish mechanisms is defined by the client instance. This URI is opaque to the AS, but can contain information relevant to the client instance's operations. In particular, the client instance can include state information to allow the callback request to be associated with an ongoing grant request.

Since this URI is exposed to the end user's browser, it is susceptible to both logging and manipulation in transit before the request is made to the client software. As such, a client instance should never put security-critical or private information into the callback URI in a cleartext form. For example, if the client software includes a post-redirect target URI in its callback URI to the AS, this target URI could be manipulated by an attacker, creating an open redirector at the client. Instead, a client instance can use an unguessable identifier into the URI that can then be used by the client software to look up the details of the pending request. Since this approach requires some form of statefulness by the client software during the redirection process, clients that are not capable of holding state through a redirect should not use redirect-based interaction mechanisms.

12.16. Redirection Status Codes

As already described in [I-D.ietf-oauth-security-topics], a server should never use the HTTP 307 status code to redirect a request that potentially contains user credentials. If an HTTP redirect is used for such a request, the HTTP status code 303 "See Other" should be used instead.

The status code 307, as defined in the HTTP standard [RFC7231], requires the user agent to preserve the method and body of a request, thus submitting the body of the POST request to the redirect target. In the HTTP standard [RFC7231], only the status code 303 unambiguously enforces rewriting the HTTP POST request to an HTTP GET request, which eliminates the POST body from the redirected request. For all other status codes, including status code 302, user agents are allowed not to rewrite a POST request into a GET request and thus to resubmit the body.

The use of status code 307 results in a vulnerability when using the redirect interaction finish method (Section 3.3.5). With this method, the AS potentially prompts the RO to enter their credentials in a form that is then submitted back to the AS (using an HTTP POST request). The AS checks the credentials and, if successful, may directly redirect the RO to the client instance's redirect URI. Due to the use of status code 307, the RO's user agent now transmits the RO's credentials to the client instance. A malicious client instance can then use the obtained credentials to impersonate the RO at the AS.

Redirection away from the initial URI in an interaction session could also leak information found in that initial URI through the HTTP Referer header field, which would be sent by the user agent to the redirect target. To avoid such leakage, a server can first redirect to an internal interstitial page without any identifying or sensitive information on the URI before processing the request. When the user agent is ultimately redirected from this page, no part of the original interaction URI will be found in the Referrer header.

12.17. MTLS Message Integrity

The MTLS key proofing mechanism (Section 7.3.2) provides a means for a client instance to present a key using a certificate at the TLS layer. Since TLS protects the entire HTTP message in transit, verification of the TLS client certificate presented with the message provides a sufficient binding between the two. However, since TLS is functioning at a separate layer from HTTP, there is no direct connection between the TLS key presentation and the message itself, other than the fact that the message was presented over the TLS channel. That is to say, any HTTP message can be presented over the TLS channel in question with the same level of trust. The verifier is responsible for ensuring the key in the TLS client certificate is the one expected for a particular request. For example, if the request is a grant request (Section 2), the AS needs to compare the TLS client certificate presented at the TLS layer to the key identified in the request body itself (either by value or through a referenced identifier).

Furthermore, the prevalence of the TLS-terminating reverse proxy (TTRP) pattern in deployments adds a wrinkle to the situation. In this common pattern, the TTRP validates the TLS connection and then forwards the HTTP message contents onward to an internal system for processing. The system processing the HTTP message no longer has access to the original TLS connection's information and context. To compensate for this, the TTRP could inject the TLS client certificate into the forwarded request as a header parameter using [I-D.ietf-httpbis-client-cert-field], giving the downstream system access to the certificate information. The TTRP has to be trusted to provide accurate certificate information, and the connection between the TTRP and the downstream system also has to be protected. The TTRP could provide some additional assurance, for example, by adding its own signature to the Client-Cert header field using [I-D.ietf-httpbis-message-signatures]. This signature would be effectively ignored by GNAP but understood by the downstream service as part of its deployment.

Additional considerations for different types of deployment patterns and key distribution mechanisms for MTLS are found in Section 12.18.

12.18. MTLS Deployment Patterns

GNAP does not specify how a client instance's keys could be made known to the AS ahead of time. Public Key Infrastructure (PKI) can be used to manage the keys used by client instances when calling the AS, allowing the AS to trust a root key from a trusted authority. This method is particularly relevant to the MTLS key proofing method, where the client instance presents its certificate to the AS as part of the TLS connection. An AS using PKI to validate the MTLS connection would need to ensure that the presented certificate was issued by a trusted certificate authority before allowing the connection to continue. PKI-based certificates would allow a key to be revoked and rotated through management at the certificate authority without requiring additional registration or management at the AS. PKI has historically been difficult to deploy, especially at scale, but it remains an appropriate solution for systems where the required overhead is not an impediment.

MTLS in GNAP need not use a PKI backing, as self-signed certificates and certificates from untrusted authorities can still be presented as part of a TLS connection. In this case, the verifier would validate the connection but accept whatever certificate was presented by the client software. This specific certificate would then be bound to all future connections from that client software by being bound to the resulting access tokens. See Section 12.17 for more considerations on MTLS as a key proofing mechanism.

12.19. Interception of Responses from the AS

Responses from the AS contain information vital to both the security and privacy operations of GNAP. This information includes nonces used in cryptographic calculations, subject identifiers, assertions, public keys, and information about what client software is requesting and was granted.

In addition, if bearer tokens are used or keys are issued alongside a bound access token, the response from the AS contains all information necessary for use of the contained access token. Any party that is capable of viewing such a response, such as an intermediary proxy, would be able to exfiltrate and use this token. If the access token is instead bound to the client instance's presented key, intermediaries no longer have sufficient information to use the token. They can still, however, gain information about the end user as well as the actions of the client software.

12.20. Key Distribution

The keys for client instances could be distributed as part of the deployment process of instances of the client software. For example, an application installation framework could generate a keypair for each copy of client software, then both install it into the client software upon installation and registering that instance with the AS.

Additionally, it's possible for the AS to generate keys to be used with access tokens that are separate from the keys used by the client instance to request tokens. In this method, the AS would generate the asymmetric keypair or symmetric key and return the entire key, including all private signing information, to the client instance alongside the access token itself. This approach would make interception of the return from the token endpoint equivalent to that of a bearer token, since all information required to use the access token would be present in the request.

12.21. Interaction Finish Modes and Polling

During the interaction process, the client instance usually hands control of the user experience over to another component, beit the system browser, another application, or some action the resource owner is instructed to take on another device. By using an interaction finish method, the client instance can be securely notified by the AS when the interaction is completed and the next phase of the protocol should occur. This process includes information that the client instance can use to validate the finish call from the AS and prevent some injection, session hijacking, and phishing attacks.

Some types of client deployment are unable to receive an interaction finish message. Without an interaction finish method to notify it, the client instance will need to poll the grant continuation API while waiting for the resource owner to approve or deny the request. An attacker could take advantage of this situation by capturing the interaction start parameters and phishing a legitimate user into authorizing the attacker's waiting client instance, which would in turn have no way of associating the completed interaction with the start of the request.

However, it is important to note that this pattern is practically indistinguishable from some legitimate use cases. For example, a smart device emits a code for the resource owner to enter on a separate device. The smart device has to poll because the expected behavior is that the interaction will take place on the separate device, without a way to return information to the original device's context.

As such, developers need to weigh the risks of forgoing an interaction finish method against the deployment capabilities of the client software and its environment. Due to the increased security, an interaction finish method should be employed whenever possible.

12.22. Session Management for Interaction Finish Methods

When using an interaction finish method such as redirect or push, the client instance receives an unsolicited HTTP request from an unknown party. The client instance needs to be able to successfully associate this incoming request with a specific pending grant request being managed by the client instance. If the client instance is not careful and precise about this, an attacker could associate their own session at the client instance with a stolen interaction response. The means of preventing this varies by the type of client software and interaction methods in use. Some common patterns are enumerated here.

If the end user interacts with the client instance through a web browser and the redirect interaction finish method is used, the client instance can ensure that the incoming HTTP request from the finish method is presented in the same browser session that the grant request was started in. This technique is particularly useful when the redirect interaction start mode is used as well, since in many cases the end user will follow the redirection with the same browser that they are using to interact with the client instance. The client instance can then store the relevant pending grant information in the session, either in the browser storage directly (such as with a single-page application) or in an associated session store on a back-end server. In both cases, when the incoming request reaches the client instance, the session information can be used to ensure that the same party that started the request is present as the request finishes.

Ensuring that the same party that started a request is present when that request finishes can prevent phishing attacks, where an attacker starts a request at an honest client instance and tricks an honest RO into authorizing it. For example, if an honest end user (that also acts as the RO) wants to start a request through a client instance controlled by the attacker, the attacker can start a request at an honest client instance and then redirect the honest end user to the interaction URI from the attackers session with the honest client instance. If the honest end user then fails to realize that it is not authorizing the attacker-controlled client instance (with which it started its request) but the honest client instance when interacting with the AS, the attacker's session with the honest client instance would be authorized. This would give the attacker access to the honest end user's resources that the honest client instance is authorized to access. However, if after the interaction the AS redirects the honest end user back to the client instance whose grant request the end user just authorized, the honest end user is redirected to the honest client instance. The honest client instance can then detect that it is not the party that started the request that is present, since the request at the honest client instance was started by the attacker, which can prevent the attack. This is related to Section 12.13, because again the attack can be prevented by the AS informing the user as much as possible about the client instance that is to be authorized.

If the end user does not interact with the client instance through a web browser or the interaction start method does not use the same browser or device that the end user is interacting through (such as the launch of a second device through a scannable code or presentation of a user code) the client instance will not be able to strongly associate an incoming HTTP request with an established session with the end user. This is also true when the push interaction finish method is used, since the HTTP request comes directly from the interaction component of the AS. In these circumstances, the client instance can at least ensure that the incoming HTTP request can be uniquely associated with an ongoing grant request by making the interaction finish callback URI unique for the grant when making the interaction request (Section 2.5.2). Mobile applications and other client instances that generally serve only a single end user at a time can use this unique incoming URL to differentiate between a legitimate incoming request and an attacker's stolen request.

If the client instance does not have the ability to use an interaction finish method, it can use polling to continue the request. The tradeoffs of this approach are discussed in Section 12.21, and if possible, an explicit interaction finish method should be used instead.

12.23. Storage of Information During Interaction and Continuation

When starting an interactive grant request, a client application has a number of protocol elements that it needs to manage, including nonces, references, keys, access tokens, and other elements. During the interaction process, the client instance usually hands control of the user experience over to another component, beit the system browser, another application, or some action the resource owner is instructed to take on another device. In order for the client instance to make its continuation call, it will need to recall all of these protocol elements. Usually this means the client instance will need to store these protocol elements in some retrievable fashion.

If the security protocol elements are stored on the end user's device, such as in browser storage or in local application data stores, capture and exfiltration of this information could allow an attacker to continue a pending transaction instead of the client instance. Client software can make use of secure storage mechanisms, including hardware-based key and data storage, to prevent such exfiltration.

Note that in GNAP, the client instance has to choose its interaction finish URI prior to making the first call to the AS. As such, the interaction finish URI will often have a unique identifier for the ongoing request, allowing the client instance to access the correct portion of its storage. Since this URI is passed to other parties and often used through a browser, this URI should not contain any security-sensitive information that would be valuable to an attacker, such as any token identifier, nonce, or user information. Instead, a cryptographically random value is suggested.

12.24. Denial of Service (DoS) through Grant Continuation

When a client instance starts off an interactive process, it will eventually need to continue the grant request in a subsequent message to the AS. It's possible for a naive client implementation to continuously send continuation requests to the AS while waiting for approval, especially if no interaction finish method is used. Such constant requests could overwhelm the AS's ability to respond to both these and other requests.

To mitigate this for well-behaved client software, the continuation response contains a wait parameter that is intended to tell the client instance how long it should wait until making its next request. This value can be used to back off client software that is checking too quickly by returning increasing wait times for a single client instance.

If client software ignores the wait value and makes its continuation calls too quickly, or if the client software assumes the absence of the wait values means it should poll immediately, the AS can choose to return errors to the offending client instance, including possibly canceling the ongoing grant request. With well-meaning client software these errors can indicate a need to change the client software's programmed behavior.

12.25. Exhaustion of Random Value Space

Several parts of the GNAP process make use of unguessable randomized values, such as nonces, tokens, and randomized URIs. Since these values are intended to be unique, a sufficiently powerful attacker could make a large number of requests to trigger generation of randomized values in an attempt to exhaust the random number generation space. While this attack is particularly applicable to the AS, client software could likewise be targeted by an attacker triggering new grant requests against an AS.

To mitigate this, software can ensure that its random values are chosen from a significantly large pool that exhaustion of that pool is prohibitive for an attacker. Additionally, the random values can be time-boxed in such a way as their validity windows are reasonably short. Since many of the random values used within GNAP are used within limited portions of the protocol, it is reasonable for a particular random value to be valid for only a small amount of time. For example, the nonces used for interaction finish hash calculation need only to be valid while the client instance is waiting for the finish callback and can be functionally expired when the interaction has completed. Similarly, artifacts like access tokens and the interaction reference can be limited to have lifetimes tied to their functional utility. Finally, each different category of artifact (nonce, token, reference, identifier, etc.) can be generated from a separate random pool of values instead of a single global value space.

12.26. Front-channel URIs

Some interaction methods in GNAP make use of URIs accessed through the end user's browser, known collectively as front-channel communication. These URIs are most notably present in the redirect interaction start method and the redirect interaction finish mode. Since these URIs are intended to be given to the end user, the end user and their browser will be subjected to anything hosted at that URI including viruses, malware, and phishing scams. This kind of risk is inherent to all redirection-based protocols, including GNAP when used in this way.

When talking to a new or unknown AS, a client instance might want to check the URI from the interaction start against a blocklist and warn the end user before redirecting them. Many client instances will provide an interstitial message prior to redirection in order to prepare the user for control of the user experience being handed to the domain of the AS, and such a method could be used to warn the user of potential threats. For instance, a rogue AS impersonating a well-known service provider. Client software can also prevent this by managing an allowlist of known and trusted AS's.

Alternatively, an attacker could start a GNAP request with a known and trusted AS but include their own attack site URI as the callback for the redirect finish method. The attacker would then send the interaction start URI to the victim and get them to click on it. Since the URI is at the known AS, the victim is inclined to do so. The victim will then be prompted to approve the attacker's application, and in most circumstances the victim will then be redirected to the attacker's site whether or not the user approved the request. The AS could mitigate this partially by using a blocklist and allowlist of interaction finish URIs during the client instance's initial request, but this approach can be especially difficult if the URI has any dynamic portion chosen by the client software. The AS can couple these checks with policies associated with the client instance that has been authenticated in the request. If the AS has any doubt about the interaction finish URI, the AS can provide an interstitial warning to the end user before processing the redirect.

Ultimately, all protocols that use redirect-based communication through the user's browser are susceptible to having an attacker try to co-opt one or more of those URIs in order to harm the user. It is the responsibility of the AS and the client software to provide appropriate warnings, education, and mitigation to protect end users.

12.27. Processing Assertions

Identity assertions can be used in GNAP to convey subject information, both from the AS to the client instance in a response (Section 3.4) and from the client instance to the AS in a request (Section 2.2). In both of these circumstances, when an assertion is passed in GNAP, the receiver of the assertion needs to parse and process the assertion. As assertions are complex artifacts with their own syntax and security, special care needs to be taken to prevent the assertion values from being used as an attack vector.

All assertion processing needs to account for the security aspects of the assertion format in use. In particular, the processor needs to parse the assertion from a JSON string object, and apply the appropriate cryptographic processes to ensure the integrity of the assertion.

For example, when SAML 2 assertions are used, the receiver hast to parse an XML document. There are many well-known security vulnerabilities in XML parsers, and the XML standard itself can be attacked through the use of processing instructions and entity expansions to cause problems with the processor. Therefore, any system capable of processing SAML 2 assertions also needs to have a secure and correct XML parser. In addition to this, the SAML 2 specification uses XML Signatures, which have their own implementation problems that need to be accounted for. Similar requirements exist for OpenID Connect's ID token, which is based on the JSON Web Token (JWT) format and the related JSON Object Signing And Encryption (JOSE) cryptography suite.

12.28. Stolen Token Replay

If a client instance can request tokens at multiple AS's, and the client instance uses the same keys to make its requests across those different AS's, then it is possible for an attacker to replay a stolen token issued by an honest AS from a compromised AS, thereby binding the stolen token to the client instance's key in a different context. The attacker can manipulate the client instance into using the stolen token at an RS, particularly at an RS that is expecting a token from the honest AS. Since the honest AS issued the token and the client instance presents the token with its expected bound key, the attack succeeds.

This attack has several preconditions. In this attack, the attacker does not need access to the client instance's key and cannot use the stolen token directly at the RS, but the attacker is able to get the access token value in some fashion. The client instance also needs to be configured to talk to multiple AS's, including the attacker's controlled AS. Finally, the client instance needs to be able to be manipulated by the attacker to call the RS while using a token issued from the stolen AS. The RS does not need to be compromised or made to trust the attacker's AS.

To protect against this attack, the client instance can use a different key for each AS that it talks to. Since the replayed token will be bound to the key used at the honest AS, the uncompromised RS will reject the call since the client instance will be using the key used at the attacker's AS instead with the same token. When the MTLS key proofing method is used, a client instance can use self-signed certificates to use a different key for each AS that it talks to, as discussed in Section 12.18.

Additionally, the client instance can keep a strong association between the RS and a specific AS that it trusts to issue tokens for that RS. This strong binding also helps against some forms of AS mix-up attacks (Section 12.10). Managing this binding is outside the scope of GNAP core, but it can be managed either as a configuration element for the client instance or dynamically through discovering the AS from the RS (Section 9.1).

The details of this attack are available in [HELMSCHMIDT2022] with additional discussion and considerations.

12.29. Self-contained Stateless Access Tokens

The contents and format of the access token are at the discretion of the AS, and are opaque to the client instance within GNAP. As discussed in the companion document, [I-D.ietf-gnap-resource-servers], the AS and RS can make use of stateless access tokens with an internal structure and format. These access tokens allow an RS to validate the token without having to make any external calls at runtime, allowing for benefits in some deployments, the discussion of which are outside the scope of this specification.

However, the use of such self-contained access tokens has an effect on the ability of the AS to provide certain functionality defined within this specification. Specifically, since the access token is self-contained, it is difficult or impossible for an AS to signal to all RS's within an ecosystem when a specific access token has been revoked. Therefore, an AS in such an ecosystem should probably not offer token revocation functionality to client instances, since the client instance's calls to such an endpoint is effectively meaningless. However, a client instance calling the token revocation function will also throw out its copy of the token, so such a placebo endpoint might not be completely meaningless. Token rotation similarly difficult because the AS has to revoke the old access token after a rotation call has been made. If the access tokens are completely self-contained and non-revocable, this means that there will be a period of time during which both the old and new access tokens are valid and usable, which is an increased security risk for the environment.

These problems can be mitigated by keeping the validity time windows of self-contained access tokens reasonably short, limiting the time after a revocation event that a revoked token could be used. Additionally, the AS could proactively signal to RS's under its control identifiers for revoked tokens that have yet to expire. This type of information push would be expected to be relatively small and infrequent, and its implementation is outside the scope of this specification.

12.30. Network Problems and Token and Grant Management

If a client instance makes a call to rotate an access token but the network connection is dropped before the client instance receives the response with the new access token, the system as a whole can end up in an inconsistent state, where the AS has already rotated the old access token and invalidated it, but the client instance only has access to the invalidated access token and not the newly rotated token value. If the client instance retries the rotation request, it would fail because the client is no longer presenting a valid and current access token. A similar situation can occur during grant continuation, where the same client instance calls to continue or update a grant request without successfully receiving the results of the update.

To combat this, both grant Management (Section 5) and token management (Section 6) are designed to be idempotent, where subsequent calls to the same function with the same credentials are meant to produce the same results. For example, multiple calls to rotate the same access token need to result in the same rotated token value.

In practice, an AS can hold on to an old token value for such limited purposes. For example, to support rotating access tokens over unreliable networks, the AS receives the initial request to rotate an access token and creates a new token value and returns it. The AS also marks the old token value as having been used to create the newly-rotated token value. If the AS sees the old token value within a small enough time window, such as a few seconds since the first rotation attempt, the AS can return the same rotated access token. Furthermore, once the system has seen the newly-rotated token in use, the original token can be discarded because the client instance has proved that it did receive the token. The result of this is a system that is eventually self-consistent without placing an undue complexity burden on the client instance.

12.31. Server-side Request Forgery (SSRF)

There are several places within GNAP where a URI can be given to a party causing it to fetch that URI during normal operation of the protocol. If an attacker is able to control the value of one of these URIs within the protocol, the attacker could cause the target system to execute a request on a URI that is within reach of the target system but normally unavailable to the attacker. For example, an attacker sending a URL of http://localhost/admin to cause the server to access an internal function on itself, or https://192.168.0.14/ to call a service behind a firewall. Even if the attacker does not gain access to the results of the call, the side effects of such requests coming from a trusted host can be problematic to the security and sanctity of such otherwise unexposed endpoints.

In GNAP, the most vulnerable place in the core protocol is the push-based post-interaction finish method (Section 4.2.2), as the client instance is less trusted than the AS and can use this method to make the AS call an arbitrary URI. While it is not required by the protocol, the AS can fetch other client-instance provided URIs such as the logo image or home page, for verification or privacy-preserving purposes before displaying them to the resource owner as part of a consent screen. Furthermore, extensions to GNAP that allow or require URI fetch could also be similarly susceptible, such as a system for having the AS fetch a client instance's keys from a presented URI instead of the client instance presenting the key by value. Such extensions are outside the scope of this specification, but any system deploying such an extension would need to be aware of this issue.

To help mitigate this problem, similar approaches to protecting parties against malicious redirects (Section 12.26) can be used. For example, all URIs that can result in a direct request being made by a party in the protocol can be filtered through an allowlist or blocklist. For example, an AS that supports the push based interaction finish can compare the callback URI in the interaction request to a known URI for a pre-registered client instance, or it can ensure that the URI is not on a blocklist of sensitive URLs such as internal network addresses. However, note that because these types of calls happen outside of the view of human interaction, it is not usually feasible to provide notification and warning to someone before the request needs to be executed, as is the case with redirection URLs. As such, SSRF is somewhat more difficult to manage at runtime, and systems should generally refuse to fetch a URI if unsure.

13. Privacy Considerations

The privacy considerations in this section are modeled after the list of privacy threats in [RFC6973], "Privacy Considerations for Internet Protocols", and either explain how these threats are mitigated or advise how the threats relate to GNAP.

13.1. Surveillance

Surveillance is the observation or monitoring of an individual's communications or activities. Surveillance can be conducted by observers or eavesdroppers at any point along the communications path.

GNAP assumes the TLS protection used throughout the spec is intact. Without the protection of TLS, there are many points throughout the use of GNAP that would lead to possible surveillance.

13.1.1. Surveillance by the Client

The purpose of GNAP is to authorize clients to be able to access information on behalf of a user. So while it is expected that the client may be aware of the user's identity as well as data being fetched for that user, in some cases the extent of the client may be beyond what the user is aware of. For example, a client may be implemented as multiple distinct pieces of software, such as a logging service or a mobile app that reports usage data to an external backend service.

13.1.2. Surveillance by the Authorization Server

The role of the authorization server is to manage the authorization of client instances to protect access to the user's data. In this role, the authorization server is by definition aware of each authorization of a client instance by a user. When the authorization server shares user information with the client instance, it needs to make sure that it has the permission from that user to do so.

Additionally, as part of the authorization grant process, the authorization server may be aware of which resource servers the client intends to use an access token at. However, it is possible to design a system using GNAP in which this knowledge is not made available to the authorization server, such as by avoiding the use of the locations object in the authorization request.

If the authorization server's implementation of access tokens is such that it requires a resource server call back to the authorization server to validate them, then the authorization server will be aware of which resource servers are actively in use and by which users and which clients. To avoid this possibility, the authorization server would need to structure access tokens in such a way that they can be validated by the resource server without notifying the authorization server that the token is being validated.

13.2. Stored Data

Several parties in the GNAP process are expected to persist data at least temporarily, if not semi-permanently, for the normal functioning of the system. If compromised, this could lead to exposure of sensitive information. This section documents the potentially sensitive information each party in GNAP is expected to store for normal operation. Naturally it is possible that any party is storing information for longer than technically necessary of the protocol mechanics (such as audit logs, etc).

The authorization server is expected to store subject identifiers for users indefinitely, in order to be able to include them in the responses to clients. The authorization server is also expected to store client key identifiers associated with display information about the client such as its name and logo.

The client is expected to store its client instance key indefinitely, in order to authenticate to the authorization server for the normal functioning of the GNAP flows. Additionally, the client will be temporarily storing artifacts issued by the authorization server during a flow, and these artifacts SHOULD be discarded by the client when the transaction is complete.

The resource server is not required to store any state for its normal operation. Depending on the implementation of access tokens, the resource server may need to cache public keys from the authorization server in order to validate access tokens.

13.3. Intrusion

Intrusion refers to the ability of various parties to send unsolicited messages or cause denial of service for unrelated parties.

If the resource owner is different from the end user, there is an opportunity for the end user to cause unsolicited messages to be sent to the resource owner if the system prompts the resource owner for consent when an end user attempts to access their data.

The format and contents of subject identifiers are intentionally not defined by GNAP. If the authorization server uses values for subject identifiers that are also identifiers for communication channels, (e.g. an email address or phone number), this opens up the possibility for a client to learn this information when it was not otherwise authorized to access this kind of data about the user.

13.4. Correlation

The threat of correlation is the combination of various pieces of information related to an individual in a way that defies their expectations of what others know about them.

13.4.1. Correlation by Clients

The biggest risk of correlation in GNAP is when an authorization server returns stable consistent user identifiers to multiple different applications. In this case, applications created by different parties would be able to correlate these user identifiers out of band in order to know which users they have in common.

The most common example of this in practice is tracking for advertising purposes, such that client A shares their list of user IDs with an ad platform that is then able to retarget ads to applications created by other parties. In contrast, a positive example of correlation is a corporate acquisition where two previously unrelated clients now do need to be able to identify the same user between the two clients.

13.4.2. Correlation by Resource Servers

Unrelated resource servers also have an opportunity to correlate users if the authorization server includes stable user identifiers in access tokens or in access token introspection responses.

In some cases a resource server may not actually need to be able to identify users, (such as a resource server providing access to a company cafeteria menu which only needs to validate whether the user is a current employee), so authorization servers should be thoughtful of when user identifiers are actually necessary to communicate to resource servers for the functioning of the system.

However, note that the lack of inclusion of a user identifier in an access token may be a risk if there is a concern that two users may voluntarily share access tokens between them in order to access protected resources. For example, if a website wants to limit access to only people over 18, and such does not need to know any user identifiers, an access token may be issued by an AS contains only the claim "over 18". If the user is aware that this access token doesn't reference them individually, they may be willing to share the access token with a user who is under 18 in order to let them get access to the website. (Note that the binding of an access token to a non-extractable client instance key also prevents the access token from being voluntarily shared.)

13.4.3. Correlation by Authorization Servers

Clients are expected to be identified by their client instance key. If a particular client instance key is used at more than one authorization server, this could open up the possibility for multiple unrelated authorization servers to correlate client instances. This is especially a problem in the common case where a client instance is used by a single individual, as it would allow the authorization servers to correlate that individual between them. If this is a concern of a client, the client should use distinct keys with each authorization server.

13.5. Disclosure in Shared References

Throughout many parts of GNAP, the parties pass shared references between each other, sometimes in place of the values themselves. For example the interact_ref value used throughout the flow. These references are intended to be random strings and should not contain any private or sensitive data that would potentially leak information between parties.

14. References

14.1. Normative References

[BCP195]   Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", May 2015, <https://www.rfc-editor.org/info/bcp195>.

[I-D.ietf-gnap-resource-servers]
Richer, J., Parecki, A., and F. Imbault, "Grant Negotiation and Authorization Protocol Resource Server Connections", Work in Progress, Internet-Draft, draft-ietf-gnap-resource-servers-01, 12 July 2021, <https://www.ietf.org/archive/id/draft-ietf-gnap-resource-servers-01.txt>.

[I-D.ietf-httpbis-digest-headers]
Polli, R. and L. Pardue, "Digest Fields", Work in Progress, Internet-Draft, draft-ietf-httpbis-digest-headers-07, 16 November 2021, <https://www.ietf.org/archive/id/draft-ietf-httpbis-digest-headers-07.txt>.

[I-D.ietf-httpbis-message-signatures]
Backman, A., Richer, J., and M. Sporny, "HTTP Message Signatures", Work in Progress, Internet-Draft, draft-ietf-httpbis-message-signatures-09, 6 March 2022, <https://www.ietf.org/archive/id/draft-ietf-httpbis-message-signatures-09.txt>.

[I-D.ietf-oauth-rar]
Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-10, 26 January 2022, <https://www.ietf.org/archive/id/draft-ietf-oauth-rar-10.txt>.

[I-D.ietf-secevent-subject-identifiers]
Backman, A. and M. Scurtescu, "Subject Identifiers for Security Event Tokens", Work in Progress, Internet-Draft, draft-ietf-secevent-subject-identifiers-09, 25 February 2022, <https://www.ietf.org/archive/id/draft-ietf-secevent-subject-identifiers-09.txt>.

[OIDC]   Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2397] Masinter, L., "The "data" URL scheme", RFC 2397, DOI 10.17487/RFC2397, August 1998, <https://www.rfc-editor.org/info/rfc2397>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

14.2. Informative References

[AXELAND2021]
Axeland, Å. and O. Oueidat, "Security Analysis of Attack Surfaces on the Grant Negotiation and Authorization Protocol", 2021, <https://odr.chalmers.se/handle/20.500.12380/304105>.

[HELMSCHMIDT2022]
Helmschmidt, F., "tbd", 2022, <tbd>.

[I-D.ietf-httpbis-client-cert-field]
Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", Work in Progress, Internet-Draft, draft-ietf-httpbis-client-cert-field-01, 25 January 2022, <https://www.ietf.org/archive/id/draft-ietf-httpbis-client-cert-field-01.txt>.

[I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-security-topics-19, 16 December 2021, <https://www.ietf.org/archive/id/draft-ietf-oauth-security-topics-19.txt>.

[promise-theory]
Burgess, M. and J. Bergstra, "Promise theory", January 2014, <http://markburgess.org/promises.html>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

Appendix A. Document History

* -09

  - Added security considerations on redirection status codes.

  - Added security considerations on cuckoo token attack.

  - Made token management URL required on token rotation.

  - Added considerations on token rotation and self-contained tokens.

  - Added security considerations for SSRF.

  - Moved normative requirements about end user presence to security considerations.

  - Clarified default wait times for continuation requests (including polling).

  - Clarified URI vs. URL.

  - Added "user_code_uri" mode, removed "uri" from "user_code" mode.

  - Consistently formatted all parameter lists.

  - Updated examples for HTTP Signatures.

* -08

  - Update definition for "Client" to account for the case of no end user.

  - Change definition for "Subject".

  - Expanded security and privacy considerations for more situations.

  - Added cross-links from security and privacy considerations.

  - Editorial updates.

* -07

  - Replace user handle by opaque identifier

  - Added trust relationships

  - Added privacy considerations section

  - Added security considerations.

* -06

  - Removed "capabilities" and "existing_grant" protocol fields.

  - Removed separate "instance_id" field.

  - Split "interaction_methods_supported" into "interaction_start_modes_supported" and "interaction_finish_methods_supported".

  - Added AS endpoint to hash calculation to fix mix-up attack.

  - Added "privileges" field to resource access request object.

  - Moved client-facing RS response back from GNAP-RS document.

  - Removed oauthpop key binding.

  - Removed dpop key binding.

  - Added example DID identifier.

  - Changed token response booleans to flag structure to match request.

   - Updated signature examples to use HTTP Message Signatures.

* -05

   - Changed "interaction_methods" to "interaction_methods_supported".

   - Changed "key_proofs" to "key_proofs_supported".

   - Changed "assertions" to "assertions_supported".

   - Updated discovery and field names for subject formats.

   - Add an appendix to provide protocol rationale, compared to OAuth2.

   - Updated subject information definition.

   - Refactored the RS-centric components into a new document.

   - Updated cryptographic proof of possession methods to match current reference syntax.

   - Updated proofing language to use "signer" and "verifier" generically.

   - Updated cryptographic proof of possession examples.

   - Editorial cleanup and fixes.

   - Diagram cleanup and fixes.

* -04

   - Updated terminology.

   - Refactored key presentation and binding.

   - Refactored "interact" request to group start and end modes.

   - Changed access token request and response syntax.

   - Changed DPoP digest field to 'htd' to match proposed FAPI profile.

   - Include the access token hash in the DPoP message.

   - Removed closed issue links.

   - Removed function to read state of grant request by client.

   - Closed issues related to reading and updating access tokens.

* -03

   - Changed "resource client" terminology to separate "client instance" and "client software".

   - Removed OpenID Connect "claims" parameter.

   - Dropped "short URI" redirect.

   - Access token is mandatory for continuation.

   - Removed closed issue links.

   - Editorial fixes.

* -02

   - Moved all "editor's note" items to GitHub Issues.

   - Added JSON types to fields.

   - Changed "GNAP Protocol" to "GNAP".

   - Editorial fixes.

* -01

   - "updated_at" subject info timestamp now in ISO 8601 string format.

   - Editorial fixes.

   - Added Aaron and Fabien as document authors.

* -00

   - Initial working group draft.

Appendix B. Compared to OAuth 2.0

GNAP's protocol design differs from OAuth 2.0's in several fundamental ways:

1. *Consent and authorization flexibility:*

OAuth 2.0 generally assumes the user has access to the a web browser. The type of interaction available is fixed by the grant type, and the most common interactive grant types start in the browser. OAuth 2.0 assumes that the user using the client software is the same user that will interact with the AS to approve access.

GNAP allows various patterns to manage authorizations and consents required to fulfill this requested delegation, including information sent by the client instance, information supplied by external parties, and information gathered through the interaction process. GNAP allows a client instance to list different ways that it can start and finish an interaction, and these can be mixed together as needed for different use cases. GNAP interactions can use a browser, but don't have to. Methods can use inter-application messaging protocols, out-of-band data transfer, or anything else. GNAP allows extensions to define new ways to start and finish an interaction, as new methods and platforms are expected to become available over time. GNAP is designed to allow the end user and the resource owner to be two different people, but still works in the optimized case of them being the same party.

2. *Intent registration and inline negotiation:*

   OAuth 2.0 uses different "grant types" that start at different endpoints for different purposes. Many of these require discovery of several interrelated parameters.

   GNAP requests all start with the same type of request to the same endpoint at the AS. Next steps are negotiated between the client instance and AS based on software capabilities, policies surrounding requested access, and the overall context of the ongoing request. GNAP defines a continuation API that allows the client instance and AS to request and send additional information from each other over multiple steps. This continuation API uses the same access token protection that other GNAP-protected APIs use. GNAP allows discovery to optimize the requests but it isn't required thanks to the negotiation capabilities.

3. *Client instances:*

OAuth 2.0 requires all clients to be registered at the AS and to use a client_id known to the AS as part of the protocol. This client_id is generally assumed to be assigned by a trusted authority during a registration process, and OAuth places a lot of trust on the client_id as a result. Dynamic registration allows different classes of clients to get a client_id at runtime, even if they only ever use it for one request.

GNAP allows the client instance to present an unknown key to the AS and use that key to protect the ongoing request. GNAP's client instance identifier mechanism allows for pre-registered clients and dynamically registered clients to exist as an optimized case without requiring the identifier as part of the protocol at all times.

4. *Expanded delegation:*

OAuth 2.0 defines the "scope" parameter for controlling access to APIs. This parameter has been coopted to mean a number of different things in different protocols, including flags for turning special behavior on and off, including the return of data apart from the access token. The "resource" parameter and RAR extensions (as defined in [I-D.ietf-oauth-rar]) expand on the "scope" concept in similar but different ways.

GNAP defines a rich structure for requesting access, with string references as an optimization. GNAP defines methods for requesting directly-returned user information, separate from API access. This information includes identifiers for the current user and structured assertions. The core GNAP protocol makes no assumptions or demands on the format or contents of the access token, but the RS extension allows a negotiation of token formats between the AS and RS.

5. *Cryptography-based security:*

OAuth 2.0 uses shared bearer secrets, including the client_secret and access token, and advanced authentication and sender constraint have been built on after the fact in inconsistent ways.

In GNAP, all communication between the client instance and AS is bound to a key held by the client instance. GNAP uses the same cryptographic mechanisms for both authenticating the client (to the AS) and binding the access token (to the RS and the AS). GNAP allows extensions to define new cryptographic protection mechanisms, as new methods are expected to become available over time. GNAP does not have a notion of "public clients" because key information can always be sent and used dynamically.

6. *Privacy and usable security:*

   OAuth 2.0's deployment model assumes a strong binding between the AS and the RS.

   GNAP is designed to be interoperable with decentralized identity standards and to provide a human-centric authorization layer. In addition to the core protocol, GNAP supports various patterns of communication between RSs and ASs through extensions. GNAP tries to limit the odds of a consolidation to just a handful of super-popular AS services.

Appendix C. Component Data Models

While different implementations of this protocol will have different realizations of all the components and artifacts enumerated here, the nature of the protocol implies some common structures and elements for certain components. This appendix seeks to enumerate those common elements.

TBD: Client has keys, allowed requested resources, identifier(s), allowed requested subjects, allowed

TBD: AS has "grant endpoint", interaction endpoints, store of trusted client keys, policies

TBD: Token has RO, user, client, resource list, RS list,

Appendix D. Example Protocol Flows

The protocol defined in this specification provides a number of features that can be combined to solve many different kinds of authentication scenarios. This section seeks to show examples of how the protocol would be applied for different situations.

Some longer fields, particularly cryptographic information, have been truncated for display purposes in these examples.

D.1. Redirect-Based User Interaction

In this scenario, the user is the RO and has access to a web browser, and the client instance can take front-channel callbacks on the same device as the user. This combination is analogous to the OAuth 2.0 Authorization Code grant type.

The client instance initiates the request to the AS. Here the client instance identifies itself using its public key.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            {
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            }
        ],
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-1",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        }
      }
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The AS processes the request and determines that the RO needs to interact. The AS returns the following response giving the client instance the information it needs to connect. The AS has also indicated to the client instance that it can use the given instance identifier to identify itself in future requests (Section 2.3.1).

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "interact": {
      "redirect":
        "https://server.example.com/interact/4CF492MLVMSW9MKM",
      "finish": "MBDOFXG4Y5CVJCX821LH"
    }
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    },
    "instance_id": "7C7C4AZ9KHRS6X63AJAO"
}
```

The client instance saves the response and redirects the user to the interaction_url by sending the following HTTP message to the user's browser.

```
HTTP 302 Found
Location: https://server.example.com/interact/4CF492MLVMSW9MKM
```

The user's browser fetches the AS's interaction URI. The user logs in, is identified as the RO for the resource being requested, and approves the request. Since the AS has a callback parameter, the AS generates the interaction reference, calculates the hash, and redirects the user back to the client instance with these additional values added as query parameters.

NOTE: '\' line wrapping per RFC 8792

```
HTTP 302 Found
Location: https://client.example.net/return/123455\
  ?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2\
    HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

The client instance receives this request from the user's browser. The client instance ensures that this is the same user that was sent out by validating session information and retrieves the stored pending request. The client instance uses the values in this to validate the hash parameter. The client instance then calls the continuation URI and presents the handle and interaction reference in the request body. The client instance signs the request as above.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

The AS retrieves the pending request based on the handle and issues an access token and returns this to the client instance.

NOTE: '\' line wrapping per RFC 8792

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [{
            "actions": [
                "read",
                "write",
                "dolphin"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        }]
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    }
}
```

D.2. Secondary Device Interaction

In this scenario, the user does not have access to a web browser on the device and must use a secondary device to interact with the AS. The client instance can display a user code or a printable QR code. The client instance is not able to accept callbacks from the AS and needs to poll for updates while waiting for the user to authorize the request.

The client instance initiates the request to the AS.

```
   POST /tx HTTP/1.1
   Host: server.example.com
   Content-Type: application/json
   Signature-Input: sig1=...
   Signature: sig1=...
   Content-Digest: sha-256=...

   {
       "access_token": {
           "access": [
               "dolphin-metadata", "some other thing"
           ],
       },
       "client": "7C7C4AZ9KHRS6X63AJAO",
       "interact": {
           "start": ["redirect", "user_code"]
       }
   }
```

The AS processes this and determines that the RO needs to interact. The AS supports both redirect URIs and user codes for interaction, so it includes both. Since there is no interaction finish mode, the AS does not include a nonce, but does include a "wait" parameter on the continuation section because it expects the client instance to poll for results.

```
   HTTP/1.1 200 OK
   Content-Type: application/json
   Cache-Control: no-store

   {
       "interact": {
           "redirect": "https://srv.ex/MXKHQ",
           "user_code": {
               "code": "A1BC-3DFF"
           }
       },
       "continue": {
           "access_token": {
               "value": "80UPRY5NM33OMUKMKSKU"
           },
           "uri": "https://server.example.com/continue/VGJKPTKC50",
           "wait": 60
       }
   }
```

The client instance saves the response and displays the user code visually on its screen along with the static device URI. The client instance also displays the short interaction URI as a QR code to be scanned.

If the user scans the code, they are taken to the interaction endpoint and the AS looks up the current pending request based on the incoming URI. If the user instead goes to the static page and enters the code manually, the AS looks up the current pending request based on the value of the user code. In both cases, the user logs in, is identified as the RO for the resource being requested, and approves the request. Once the request has been approved, the AS displays to the user a message to return to their device.

Meanwhile, the client instance periodically polls the AS every 60 seconds at the continuation URI. The client instance signs the request using the same key and method that it did in the first request.

```
POST /continue/VGJKPTKC50 HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60 second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "G7YQT4KQQ5TZY9SLSS5E"
        },
        "uri": "https://server.example.com/continue/ATWHO4Q1WV",
        "wait": 60
    }
}
```

Note that the continuation URI and access token have been rotated since they were used by the client instance to make this call. The client instance polls the continuation URI after a 60 second timeout using this new information.

```
POST /continue/ATWHO4Q1WV HTTP/1.1
Host: server.example.com
Authorization: GNAP G7YQT4KQQ5TZY9SLSS5E
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...
```

The AS retrieves the pending request based on the URI and access token, determines that it has been approved, and issues an access token for the client to use at the RS.

NOTE: '\' line wrapping per RFC 8792

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

D.3. No User Involvement

In this scenario, the client instance is requesting access on its own behalf, with no user to interact with.

The client instance creates a request to the AS, identifying itself with its public key and using MTLS to make the request.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json

{
    "access_token": {
        "access": [
            "backend service", "nightly-routine-3"
        ],
    },
    "client": {
      "key": {
        "proof": "mtls",
        "cert#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
      }
    }
}
```

The AS processes this and determines that the client instance can ask for the requested resources and issues an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token",
        "access": [
            "backend service", "nightly-routine-3"
        ]
    }
}
```

D.4. Asynchronous Authorization

In this scenario, the client instance is requesting on behalf of a specific RO, but has no way to interact with the user. The AS can asynchronously reach out to the RO for approval in this scenario.

The client instance starts the request at the AS by requesting a set of resources. The client instance also identifies a particular user.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata",
            {
                "type": "financial-transaction",
                "actions": [
                    "withdraw"
                ],
                "identifier": "account-14-32-32-3",
                "currency": "USD"
            },
            "some other thing"
        ],
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "user": {
        "sub_ids": [ {
            "format": "opaque",
            "id": "J2G8G8O4AZ"
        } ]
  }
}
```

The AS processes this and determines that the RO needs to interact. The AS determines that it can reach the identified user asynchronously and that the identified user does have the ability to approve this request. The AS indicates to the client instance that it can poll for continuation.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

The AS reaches out to the RO and prompts them for consent. In this example, the AS has an application that it can push notifications in to for the specified account.

Meanwhile, the client instance periodically polls the AS every 60 seconds at the continuation URI.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60 second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "BI9QNW6V9W3XFJK4R02D"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

Note that the continuation handle has been rotated since it was used by the client instance to make this call. The client instance polls the continuation URI after a 60 second timeout using the new handle.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP BI9QNW6V9W3XFJK4R02D
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the handle and determines that it has been approved and it issues an access token.

NOTE: '\' line wrapping per RFC 8792

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

D.5. Applying OAuth 2.0 Scopes and Client IDs

While GNAP is not designed to be directly compatible with OAuth 2.0 [RFC6749], considerations have been made to enable the use of OAuth 2.0 concepts and constructs more smoothly within GNAP.

In this scenario, the client developer has a client_id and set of scope values from their OAuth 2.0 system and wants to apply them to the new protocol. Traditionally, the OAuth 2.0 client developer would put their client_id and scope values as parameters into a redirect request to the authorization endpoint.

NOTE: '\' line wrapping per RFC 8792

```
HTTP 302 Found
Location: https://server.example.com/authorize\
  ?client_id=7C7C4AZ9KHRS6X63AJAO\
  &scope=read%20write%20dolphin\
  &redirect_uri=https://client.example.net/return\
  &response_type=code\
  &state=123455
```

Now the developer wants to make an analogous request to the AS using GNAP. To do so, the client instance makes an HTTP POST and places the OAuth 2.0 values in the appropriate places.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Content-Digest: sha-256=...

{
    "access_token": {
        "access": [
            "read", "write", "dolphin"
        ],
        "flags": [ "bearer" ]
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return?state=123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The client_id can be used to identify the client instance's keys that it uses for authentication, the scopes represent resources that the client instance is requesting, and the redirect_uri and state value are pre-combined into a finish URI that can be unique per request. The client instance additionally creates a nonce to protect the callback, separate from the state parameter that it has added to its return URI.

From here, the protocol continues as above.

Appendix E. JSON Structures and Polymorphism

GNAP makes use of polymorphism within the JSON [RFC8259] structures used for the protocol. Each portion of this protocol is defined in terms of the JSON data type that its values can take, whether it's a string, object, array, boolean, or number. For some fields, different data types offer different descriptive capabilities and are used in different situations for the same field. Each data type provides a different syntax to express the same underlying semantic protocol element, which allows for optimization and simplification in many common cases.

Even though JSON is often used to describe strongly typed structures, JSON on its own is naturally polymorphic. In JSON, the named members of an object have no type associated with them, and any data type can be used as the value for any member. In practice, each member has a semantic type that needs to make sense to the parties creating and consuming the object. Within this protocol, each object member is defined in terms of its semantic content, and this semantic content might have expressions in different concrete data types for different specific purposes. Since each object member has exactly one value in JSON, each data type for an object member field is naturally mutually exclusive with other data types within a single JSON object.

For example, a resource request for a single access token is composed of an array of resource request descriptions while a request for multiple access tokens is composed of an object whose member values are all arrays. Both of these represent requests for access, but the difference in syntax allows the client instance and AS to differentiate between the two request types in the same request.

Another form of polymorphism in JSON comes from the fact that the values within JSON arrays need not all be of the same JSON data type. However, within this protocol, each element within the array needs to be of the same kind of semantic element for the collection to make sense, even when the data types are different from each other.

For example, each aspect of a resource request can be described using an object with multiple dimensional components, or the aspect can be requested using a string. In both cases, the resource request is being described in a way that the AS needs to interpret, but with different levels of specificity and complexity for the client instance to deal with. An API designer can provide a set of common access scopes as simple strings but still allow client software developers to specify custom access when needed for more complex APIs.

Extensions to this specification can use different data types for defined fields, but each extension needs to not only declare what the data type means, but also provide justification for the data type representing the same basic kind of thing it extends. For example, an extension declaring an "array" representation for a field would need to explain how the array represents something akin to the non-array element that it is replacing.

Authors' Addresses

Justin Richer (editor)
Bespoke Engineering
Email: ietf@justin.richer.org

URI: https://bspk.io/

Aaron Parecki
Okta
Email: aaron@parecki.com
URI: https://aaronparecki.com

Fabien Imbault
acert.io
Email: fabien.imbault@acert.io
URI: https://acert.io/